

深入NoSQL

Professional NoSQL

[印] Shashank Tiwari 著
巨成 译

人民邮电出版社
POSTS & TELECOM PRESS

Shashank Tiwari

创业者、开发者、技术作家、演讲者和导师，技术型创业公司Treasury of Ideas（www.treasuryofideas.com）的创始人。

他是一位经验丰富的软件开发者和企业家，长期关注高性能应用、分析、Web应用以及移动平台，对数据可视化和统计机器学习有着浓厚的兴趣，喜欢喝咖啡、吃甜点、骑自行车。他撰写了许多技术文章和著作，并且应邀在全球各地的技术会议上进行演讲。

访问www.treasuryofideas.com可以进一步了解他的公司和业务。他的博客地址是www.shanky.org，Twitter账号是@tshanky。



深入NoSQL

Professional NoSQL

[印] Shashank Tiwari 著
巨成 译

人民邮电出版社
北京

图书在版编目（C I P）数据

深入NoSQL / (印) 蒂瓦里著 ; 巨成译. -- 北京 :
人民邮电出版社, 2012.11
（图灵程序设计丛书）
书名原文: Professional NoSQL
ISBN 978-7-115-29638-2

Ⅰ. ①深… Ⅱ. ①蒂… ②巨… Ⅲ. ①数据库系统
Ⅳ. ①TP311.13

中国版本图书馆CIP数据核字(2012)第242448号

内 容 提 要

本书是一本全面的 NoSQL 实践指南。书中主要关注 NoSQL 的基本概念，以及使用 NoSQL 数据库的切实可行的解决方案。书中介绍了基于 MapReduce 的可伸缩处理，演示 Hadoop 用例，还有 Hive 和 Pig 这样的高层抽象。本书包含许多用例演示，同时也会讨论 Google、Amazon、Facebook、Twitter 和 LinkedIn 的可伸缩数据架构。

本书适合 NoSQL 数据库管理人员和开发人员阅读。

图灵程序设计丛书
深入NoSQL

◆ 著　　[印] Shashank Tiwari
译　　巨　成
责任编辑　朱　巍
执行编辑　李　瑛

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：19.25
字数：455千字　　2012年11月第1版
印数：1－4 000册　　2012年11月北京第1次印刷

著作权合同登记号　图字：01-2012-4003号

ISBN 978-7-115-29638-2

定价：69.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

献　　词

谨以此书献给我的父母 Suresh 和 Mandakini。

包括这本书在内，我做成的每一件事都离不开我挚爱的妻子 Caren、可爱的儿子 Ayaan 和 Ezra 的大力支持。

译 者 序

NoSQL 很有趣，因为它和高性能、分布式、海量数据这些热词有着千万缕的联系。学习使用 NoSQL 的过程中，我们一点点地拆解原来的数据库、分布式系统，在不同的场景，不同的假设下，重新审视各个部分。我还需要这块吗？那里是不是可以做些调整？我们开始仔细审视一些理所当然的事实。到了最后，就像第一次知道分子的学生，见识了原子的内部，也见识了一片新世界。你会好奇，这个新世界是多么的陌生，又是多么的熟悉。它的原理其实不新，我们都知道。虽说熟悉，可也还是陌生，那么多条路指向 NoSQL 世界的深处，你背着行李兴奋地站在路口，然而，该选哪条路呢？就选《深入 NoSQL》吧☺。

这本书涉猎广泛，举例具体，不需要你在网上到处翻查零散的文档。它的深度适当，内容容易理解，若你需要进一步了解某个主题，官方文档和源代码会是更好的选择。我建议速读，为此我也调整了许多段落的句式，希望句子短而清楚，像快节奏的鼓点，催着你马不停蹄，一气呵成。

程序员们总有好书读，新书看，这是一种幸福。对从小读写中文长大的人，中文更易读，也读得快。书翻译得好，不知不觉中就为读者省下了时间。这些年我读的不少书是译者辛勤劳动的结晶，希望我的这一本也能对你有所帮助。如果你也想着做点贡献，翻译些东西，我会诚恳地鼓励你，希望你在这个过程中也能有所收获。

巨成

2012 年 9 月 26 日

前　　言

随着用户内容的增长，所生成、处理、分析和归档的数据的规模快速增大，类型也快速增多。此外，一些新数据源也在生成大量数据，比如传感器、全球定位系统（GPS）、自动追踪器和监控系统。这些大数据集通常被称为大数据，它们给存储、分析和归档带来了新的机遇与挑战。

数据不仅仅快速增长，而且半结构化和稀疏的趋势也很明显。这样一来，预定义好 schema 和利用关系型引用的传统数据管理技术就受到了挑战。

在探索海量数据和半结构化数据相关问题的过程中，诞生了一系列新型数据库产品，其中包括列族数据库（column-oriented data store）、键/值数据库和文档数据库，这些数据库统称 NoSQL。

NoSQL 产品千变万化，特性和价值主张各有不同，因此常常难以选择。本书能帮助你理解整个 NoSQL 领域。书中展示了构建许多 NoSQL 产品的基本概念，覆盖了相对较多的 NoSQL 产品，而非单单深入介绍某一种产品。本书主要关注广度和基本概念，而不是全面覆盖每一种产品的 API。因为要介绍不少 NoSQL 产品，所以会涉及大量的比较分析。

如果不确定如何开始用 NoSQL 以及如何学习管理和分析大数据，那么你会发现本书是一本很好的入门指南和参考用书。

读者对象

本书的主要目标读者是开发者、架构师、数据库管理员以及技术项目经理，但任何理解数据库技术的人可能都会觉得本书很有帮助。

计算机专业的许多学生和研究员也会对大数据和 NoSQL 这一主题感兴趣，他们会因阅读本书而获益良多。

任何开始进行大数据分析和使用 NoSQL 的人也都能从本书中受益。

本书内容

本书首先介绍 NoSQL 的基础知识，然后逐步过渡到围绕性能调优和架构性指引的高阶概念上。我们主要关注与 NoSQL 相关的基本概念，并以许多不同的 NoSQL 产品为例来解释它们。书中包括了有关 MongoDB、CouchDB、HBase、Hypertable、Cassandra、Redis 和 BerkeleyDB 的演示和样例，此外还包括其他一些 NoSQL 产品。

NoSQL 很重要的一部分就是大数据处理。本书将介绍基于 MapReduce 的可伸缩处理，演示 Hadoop 用例，还有 Hive 和 Pig 这样的高层抽象。

第 10 章关注云 NoSQL，介绍 Amazon Web Service 和 Google App Engine 提供的平台。

本书包含许多用例演示，同时也会讨论 Google、Amazon、Facebook、Twitter 和 LinkedIn 的可伸缩数据架构。

在最后一部分中，本书会比较 NoSQL 产品，讨论不同产品在应用程序栈中共存的话题。

本书结构

本书分为四大部分：

- NoSQL 入门
- NoSQL 基础
- 熟悉 NoSQL
- 掌握 NoSQL

每部分的内容都是以前一部分为基础的。

第一部分为 NoSQL 入门，定义了 NoSQL 产品的类型，并初次介绍了用 NoSQL 存储和访问数据的几个例子。

- 第 1 章定义 NoSQL。
- 第 2 章以超经典的 `Hello World` 程序开头，介绍了几个使用 NoSQL 的例子。
- 第 3 章介绍 NoSQL 产品交互与接口。

第二部分介绍了各种 NoSQL 产品的一些基本概念。

- 第 4 章解释存储架构。
- 第 5 章和第 6 章介绍基本的数据管理，演示 CRUD 操作和查询机制。数据集随时间和使用情况而演变。
- 第 7 章探讨数据演变相关的问题。传统的关系型数据库关注利用索引来优化查询。
- 第 8 章介绍 NoSQL 下的索引。对 NoSQL 产品缺少事务支持的批评往往过头。
- 第 9 章陈清事务相关概念，以及分布式系统所面临的事务完整性挑战。

第三、四部分介绍高阶话题。

- 第 10 章介绍 Google App Engine 数据存储和 Amazon SimpleDB。很多大数据处理有赖于 MapReduce 风格的处理方式。
- 第 11 章介绍 MapReduce 的基本知识。
- 第 12 章扩展了 MapReduce 的覆盖范围，以演示 Hive 为 Hadoop MapReduce 任务提供的类 SQL 抽象。第 13 章回顾了数据库架构及内部结构。

第五部分是本书的最后一部分。第 14 章比较 NoSQL 产品；第 15 章提出了共存的想法，以及按需选择数据库的观点；第 16 章谈论可伸缩应用程序的优化。本部分的话题看似驳杂，却为实际应用 NoSQL 打下了基础。第 17 章展示一些工具和实用程序，部署 NoSQL 时用得上。

阅读必备

请参照各处代码示例安装相应的软件，安装步骤和设置说明参见附录 A。

本书约定

为了描述得更加清楚明了，在本书中我们有如下约定。

> 本格式表示针对目前讨论内容的注释、小技巧、提示等。

段落样式如下。

- ❑ 楷体表示新词及重点词。
- ❑ 文件名、URL 和书中的代码使用如下这种字体：`persistence.properties`。
- ❑ 代码有两种展示方法：大部分代码样例使用 monofont 字体，无高亮；当前上下文中重要的代码以及与之前代码段不同的代码加粗显示。

源代码

对于本书所有例子中的代码，你可以手工输入，也可以使用本书附带的源码文件。所有源码可从 www.wrox.com 下载。在网站上通过本书书名（使用检索框或书名列表）进入本书的详情页，点击下载代码的链接即可获取源码。网站上包含的代码都附有下面的图标：

代码示例的标题里包含源码文件名。如果只是代码片段，则文件名如是：

Code snippet flename

> 因为很多书名很相似，所以按 ISBN 号可能查起来更容易。本书 ISBN 号是 978-0-470-94224-6。

下载好的代码可以用解压缩工具打开。除此之外，在下载页面 www.wrox.com/dynamic/books/download.aspx 也能看到本书及 Wrox 其他图书的代码。

勘误

虽然我们尽全力消除文本和代码中的所有错误，但错误总是难以避免的。如果你发现了本书

的错误，比如拼写错误或问题代码，请反馈给我们，非常感谢！如果你能指出一个问题，也许就能避免另一位读者的困惑，同时也能帮助我们提高本书的质量。

要前往本书的勘误页，请访问 www.wrox.com，通过书名进入本书的详情页，然后点击勘误（Book Errata）链接。在该页面中，你可以看到由读者提交并由 Wrox 编者发布的本书的所有勘误。要想获得完整的图书列表及每本书的勘误链接，也可以访问 www.wrox.com/misc-pages/booklist.shtml。

如果在勘误页上没找到你发现的错误，请前往 www.wrox.com/contact/techsupport.shtml，填写表单提交你发现的错误。我们会尽快确认信息，如果勘误正确，会在勘误页上发布，并在本书的后续版本中修正问题。

P2P.WROX.COM

若想与包括本书作者在内的同行们讨论，请加入 P2P 论坛（p2p.wrox.com）。这个 Web 论坛主要用于发布 Wrox 图书信息及相关技术消息，以及与其他读者及技术用户互动。论坛提供主题订阅功能，选定了自己感兴趣的主题后，每当有相应主题的新帖发布时，论坛会以电子邮件的形式通知你。Wrox 作者、编者、其他行业专家和读者都会在这里活动。

在 p2p.wrox.com，你会发现一系列不同的论坛，它们不仅对你阅读本书有所帮助，同样也能在你开发应用时提供帮助。加入论坛请参照下面的步骤。

(1) 前往 p2p.wrox.com，点击注册链接。

(2) 阅读使用守则并点击“同意”。

(3) 输入必填信息及其他你乐于提供的可选信息，并点击“提交”。

(4) 你会收到一封邮件，其中描述了如何确认你的账户和完成加入论坛的流程。

> 阅读消息无需在 P2P 上注册，但如果要发布消息，就必须加入论坛。

加入论坛后，可以发布新消息和回复他人发布的消息。任何时候你都可以通过 Web 浏览论坛。如果希望通过电子邮件接收特定论坛的新消息，请在论坛列表中点击论坛名称旁边的“订阅本论坛”图标。

要了解更多有关 Wrox P2P 的信息，请阅读 P2P FAQ（点击 P2P 页面上的 FAQ 链接）了解论坛使用方法，以及 P2P 和 Wrox 图书常见问题。

致　谢

许多人为本书的出版付出了努力，我衷心地感谢他们。

感谢 Wiley 团队，没有他们，本书就不可能成功问世。

感谢 Matt 和 Stefan 提出宝贵的意见并进行技术审校。

感谢我的妻子和儿子们在写作过程中给予我支持和鼓励；感谢一直信任我的家人和朋友。

感谢我无意中漏掉的直接或间接为本书作出贡献的人。

——SHASHANK TIWARI

关于技术审校

STEFAN EDLICH 教授/博士是柏林 Beuth HS of Technology 的高级讲师，专攻 NoSQL、软件工程和云计算。他发表过许多科技论文和期刊文章，并且自 1993 年以来，一直积极参与各种会议和 IT 活动，就企业、NoSQL 和 ODBMS 等话题发表演讲。

此外，他还编撰了 12 本 IT 书籍，分别由 Apress、OReilly、Spektrum/Elsevier、Hanser 等出版社出版。他是 OODBMS.org e.V. 的元老，组织召开了第一届对象数据库国际会议（ICOODB.org）。他负责管理 NoSQL Archive，经常组织 NoSQL 活动，还经常撰写有关 NoSQL 的文章。

MATT INGENTHRON 是一位经验丰富的具有软件开发背景的 Web 架构师。他对构建、扩展和运维世界级的 Java、Ruby on Rails 和 AMP Web 应用很有经验。自 Couchbase 成立以来，他一直在该公司工作。他是开源 Membase NoSQL 项目的核心开发人员，是 Memcached 项目的贡献者，还负责 Java spymemcached 客户端的发展。Matt 在 NoSQL 方面涉猎广泛，包括 Hadoop、HBase 等。

目　录

第四部分　掌握 NoSQL

Part 1

第一部分

NoSQL 入门

本部分内容

第 1 章

NoSQL 的概念及适用范围

本章内容

- NoSQL 的定义
- NoSQL 的历史
- 介绍各种 NoSQL 数据库
- 列举一些流行的 NoSQL 产品

恭喜！在学习 NoSQL 的路上你已经勇敢地迈出了第一步。

与大多数新兴技术一样，NoSQL 被恐惧、不确定和疑惑笼罩着。根据对待 NoSQL 的态度，开发者大致能划分成以下三类。

- **喜欢它的人**：他们研究 NoSQL 是否适用于某套应用程序栈。他们使用 NoSQL、创造 NoSQL，一直处于 NoSQL 领域发展的前沿。
- **讨厌它的人**：这些人要么盯着 NoSQL 的短处不放，要么试图证明它毫无价值。
- **观望它的人**：此类开发者属于不可知论者，因为他们或者在等待 NoSQL 技术成熟，或者认为 NoSQL 只会流行一时，而忽略它就可以避免"炒作周期（hype cycle）"带来的负面影响，又或者他们只是没有机会使用 NoSQL。

> Gartner 发明了炒作周期一词，以表示一种技术的成熟度、市场接受度以及商业应用程度。更多内容请访问 http://en.wikipedia.org/wiki/Hype_Cycle。

我属于第一类人。写一本 NoSQL 主题的书恰好证明了我对这项技术的喜爱。喜欢/讨厌 NoSQL 的人按照信仰程度的不同还可以进一步划分：从温和派到极端主义者。我属于温和派。鉴于此，我希望展示的是，NoSQL 作为一种强有力的工具，非常适合完成某些工作，同时它也存在短处。我希望读者能带着一种开放的、没有偏见的心态来学习 NoSQL。掌握这项技术和它背后的思想，你就能对 NoSQL 的使用价值做出自己的判断，并能根据特定的应用或场景恰当地利用 NoSQL。

第 1 章为 NoSQL 入门。本章简单介绍什么是 NoSQL，它具备怎样的特性，典型用例有哪些，以及它在应用程序栈中适于哪种位置。

1.1 定义和介绍

字面上 NoSQL 是两个词的组合：No 和 SQL，它暗示了 NoSQL 技术/产品与 SQL 之间的对立性。其实这个术语的发明人和早期使用者的意思很可能是 No RDBMS（No Relational Database Management System，非关系型数据库管理系统）或者 No relational（非关系型），但因为 NoSQL 的发音更好听，最终选择了这个词。后来有人提议用 NonRel 来代替 NoSQL，还有人提出 NoSQL 实际上是"Not Only SQL"（不仅是 SQL）的简称，以试图挽回原来的术语。不管字面意思如何，今天 NoSQL 泛指这样一类数据库和数据存储，它们不遵循经典 RDBMS 原理，且常与 Web 规模的大型数据集有关。换句话说，NoSQL 并不单指一个产品或一种技术，它代表一族产品，以及一系列不同的、有时相互关联的、有关数据存储及处理的概念。

1.1.1 背景与历史

在详细介绍各种 NoSQL 类型及其相关概念之前，还有一件重要的事情，那就是了解 NoSQL 出现的大背景。非关系型数据库并不是阳春白雪，事实上，最早的非关系型存储可以追溯到第一批计算机出现的时期。在大型机出现的年代里，非关系型数据库曾获得繁荣的发展，并且在特定的专业领域里（比如存储认证和授权信息的层级目录）一直延续至今。话说回来，这些诞生在大规模互联网应用时代里，并在 NoSQL 舞台露脸的非关系型存储确实算得上是新鲜玩意儿。这些非关系型的 NoSQL 存储大都孕育自分布式和并行计算大行其道的年代。

从可被视作第一个搜索引擎的 Inktomi 开始，到后来的 Google，广泛采用的关系型数据库管理系统（RDBMS）应用于海量数据时，暴露出了一系列自身的问题。这些问题与高效的数据处理、高效的并行化、可扩展性和成本有关。在本章及之后的讨论中，我们不仅会了解所有这些问题，还会探寻可能的解决方案。

RDBMS 的挑战

Web 级别大规模数据处理对 RDBMS 的挑战其实并非特定于某一个产品，所有同类数据库都面临着严峻的考验。RDBMS 假定数据的结构已明确定义，数据是致密的，并且很大程度上是一致的。RDBMS 构建在这样的先决条件上，即数据的属性可以预先定义好，它们之间的相互关系非常稳固且被系统地引用（systematically referenced）。它还假定定义在数据上的索引能保持一致性，能统一应用以提高查询的速度。不幸的是，一旦这些假设无法成立，RDBMS 就立刻暴露出问题。当然，RDBMS 可以容忍一定程度的不规律和结构缺乏，但在松散结构的海量稀疏数据面前，RDBMS 就显得勉为其难了。在大规模数据面前，传统存储机制和访问方法捉襟见肘。为了帮助 RDBMS 扩展，你可以对表做逆规范化处理、去掉约束和放宽事务保障，但经过了这些修改的 RDBMS 其实更类似一个 NoSQL 产品。

灵活性是有代价的。NoSQL 缓解了 RDBMS 引发的问题并降低了处理海量稀疏数据的难度，但是反过来也被夺去了事务完整性的力量和灵活的索引及查询能力。极为讽刺的是，

NoSQL 产品中最令人怀念的一个特性正是 SQL，这一领域里的产品供应商们都在努力尝试各种方法弥补这一空白。

过去几年间，Google 建造了大规模可扩展的基础设施，用于支撑 Google 的搜索引擎和其他应用，包括 Google Maps、Google Earth、Gmail、Google Finance 以及 Google Apps。其策略是在应用程序栈的每个层面上分别解决问题，旨在建立一套可伸缩的基础设施来并行处理海量数据。为此 Google 创建了一整套完备的机制，包括分布式文件系统、面向列族的数据存储、分布式协调系统和基于 MapReduce 的并行算法执行环境。Google 大方地公开发布了一系列论文来解释其基础设施中一些关键的组成部分，其中最重要的论文包括下面几篇。

- Sanjay Ghemawat、Howard Gobioff 和 Shun-Tak Leung，"The Google File System"; pub.19th ACM Symposium on Operating Systems Principles, Lake George, NY, October 2003 。链接：http://labs.google.com/papers/gfs.html。
- Jeffrey Dean 和 Sanjay Ghemawat，"MapReduce: Simplified Data Processing on Large Clusters"; pub. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December 2004。链接：http://labs.google.com/papers/mapreduce.html。
- Fay Chang、Jeffrey Dean、Sanjay Ghemawat、Wilson C. Hsieh、Deborah A. Wallach、Mike Burrows、Tushar Chandra、Andrew Fikes 和 Robert E. Gruber，"Bigtable: A Distributed Storage System for Structured Data"; pub. OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November 2006。链接：http://labs.google.com/papers/bigtable.html。
- Mike Burrows，"The Chubby Lock Service for Loosely-Coupled Distributed Systems"; pub.OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November 2006。链接：http://labs.google.com/papers/chubby.html。

> 如果你在阅读这一部分或本章后面部分时，对其中介绍的一系列术语和概念感到非常困惑和不堪重负，可以稍事休息喘口气。本书采用相对轻松的节奏来解释所有相关概念。你不需要一口气学会所有东西，按顺序阅读，待读完全书，自然会理解所有与 NoSQL 和大数据相关的重要概念。

Google 相关论文的公开发表引起了开源开发者的广泛关注和浓厚兴趣。很快，第一个模仿 Google 基础设施部分特性的开源软件就开发出来了，它的创建者正是开源搜索引擎 Lucene 的发明人。紧接着，Lucene 的核心开发者们加入了 Yahoo!，在那里，依靠众多开源贡献者的支持，参照 Google 的分布式计算架构，开发者们创建出了一个能够替代 Google 基础设施所有部分的开源产品，这就是 Hadoop 及其子项目和相关项目。更多有关 Hadoop 的信息、代码和文档请访问 http://hadoop.apache.org。

在这里我们不再赘述 Hadoop 发展史，NoSQL 思想诞生在 Hadoop 发行第一个版本之前。谁在什么时候创造了 NoSQL 这个术语并不重要，重要的是 Hadoop 的出现为 NoSQL 的快速发展奠

定了坚实的基础，而 Google 的成功则帮助大众接受了新时代的分布式计算理念、Hadoop 项目以及 NoSQL。

Google 的论文激发了人们对并行大规模处理和分布式非关系型数据存储的兴趣，一年后，Amazon 分享了他们的成功经验。2007 年 Amazon 对外展示了它的分布式高可用、最终一致性数据存储，其名曰 Dynamo。更多有关 Amazon Dynamo 的内容可以在一篇研究论文里读到，论文提录如下：Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels, "Dynamo: Amazon's Highly Available Key/value Store," in the Proceedings of the 21st ACM Symposium on Operating Systems Principles, Stevenson, WA, October 2007。此外，Amazon 的 CTO Werner Vogels 在一篇博文中解释了 Amazon Dynamo 背后的关键思想，博文地址为：www.allthingsdistributed.com/2007/10/amazons_dynamo.html。

随着 NoSQL 得到两大领先 Web 巨人 Google 和 Amazon 的支持，领域中新产品相继出现。大量开发者开始在他们的应用程序中尝试这些产品。小到初创公司，大到大型企业，许多企业都开始愿意学习更多相关技术和借鉴相关方法。在不到五年的时间里，NoSQL 和用于管理大数据的概念得到了广泛传播，众多知名公司中开始涌现各种用例，其中包括 Facebook、Netflix、Yahoo、EBay、Hulu 和 IBM 等。其中许多公司也通过开源向世界贡献出了他们自己的扩展组件和新产品。

很快你将会了解到有关各种 NoSQL 产品的知识，包括它们的相似之处和不同特点，但此时此刻请允许我先暂时离题，对有关大数据及并行处理的一些挑战和解决方案做个简短介绍。此番绕行偏题旨在帮助所有读者为探索 NoSQL 产品做好准备。

1.1.2　大数据

多大的数据量才算大？如果你就这个问题询问不同的人，必然会得到不同的答案。此外，答案还可能随提问的时机而发生变化。就现在而言，任何超过几个 TB 大小的数据集都可以归为大数据。这是数据集大到开始跨越多个存储单元的典型尺寸，也是传统 RDBMS 技术开始表现出吃力的尺寸。

数据大小的计算

字节（byte）是数字信息的单位，1 字节等于 8 比特（bit）。在国际单位制中，1 个字节的每 1000 倍被赋予一个不同的名称：

Kilobyte (kB)——10^3

Megabyte (MB)——10^6

Gigabyte (GB)——10^9

Terabyte (TB)——10^{12}

Petabyte (PB)——10^{15}

Exabyte (EB)——10^{18}

Zettabyte (ZB)——10^{21}

Yottabyte (YB)——10^{24}

在传统二进制中，倍数应当是 2^{10}（或 1024）而非 10^3（或 1000）。为了避免混淆，同时还存在一个以 2 为幂的命名法：

Kibibyte (KiB)——2^{10}

Mebibyte (MiB)——2^{20}

Gibibyte (GiB)——2^{30}

Tebibyte (TiB)——2^{40}

Pebibyte (PiB)——2^{50}

Exbibyte (EiB)——2^{60}

Zebibyte (ZiB)——2^{70}

Yobibyte (YiB)——2^{80}

就在几年前，1TB 个人数据可能就算非常庞大的了，但如今，本地硬盘驱动器和备份驱动器的容量通常都有这么大。可以料想，未来几年里，即便硬盘驱动器的默认容量超过了数个 TB 也不足为奇。我们生活在一个数据大爆炸的时代里，数码相机的输出、博客、日常的社交网络更新、微博、电子文档、扫描的内容、音乐文件以及视频都在快速增多。可以说，我们在消费大量数据的同时，也在生成大量数据。

要估算电子数据或互联网的真实大小非常困难，据研究估算和数据显示，这个值非常之大，可能在 ZB 量级，甚至更大。一项正在进行的名为“The Digital Universe Decade – Are you ready?”（http://emc.com/collateral/demos/microsites/idc-digital-universe/iview.htm）的研究中，IDC 代表 EMC 展示了电子数据目前的状态及其增长速度。报告称，到 2020 年创建和复制的电子数据总规模将达到 35ZB。报告同时声称，目前生成的和可用的数据总量正在逐渐超过可用的存储总量。

其他一些值得思考的数据包括：

- ACM 里一篇于 2009 年发表的论文，标题为“MapReduce: simplified data processing on large clusters ”（ http://portal.acm.org/citation.cfm?id=1327452.1327492&coll=GUIDE&dl=&idx=J79&part=magazine&WantType=Magazines&title=Communications%20of%20the%20ACM），揭示 Google 每天要处理 24PB 的数据。
- 2009 年 Facebook 上一篇有关其相片存储系统的博文：“Needle in a haystack: efficient storage of billions of photos”（http//facebook.com/note.php?note_id=76191543919），提到 Facebook 存储的相片总量达到 1.5PB。同一篇文章还提到 Facebook 上大约存储了 600 亿张图片。
- 互联网档案的 FAQ 页面（archive.org/about/faqs.php）提到存储在互联网档案里的数据高达 2PB，并声称数据增长率达到每月 20TB。
- 电影《阿凡达》为渲染 3D CGI 效果使用的存储空间达到了 1PB（“Believe it or not: Avatar takes 1 petabyte of storage space, equivalent to a 32-year-long MP3”：http://thenextweb.com/2010/01/01/avatar-takes-1-petabyte-storagespace-equivalent-32-year-long-mp3/）。

随着数据规模的增长和数据创建来源的日趋多元化，以下挑战将日益严峻。

- 高效存储和访问大量数据很难。额外要求的容错和备份使事情变得更加复杂。
- 操作大数据集涉及大量并行进程。运行过程中要从任何故障中平稳恢复过来，同时还要在合理的时间范围内返回结果，这非常复杂。
- 各种不同数据源生成的半结构化和无结构数据的 schema 和元数据持续不断变化，对它们的管理是一个令人头疼的问题。

因此，我们需要新方法，它们在存取大量数据方面比现行方法更有效。NoSQL 和相关的大数据解决方案正是朝这个方向迈出的第一步。

除了数据，同时增长的还有规模。

磁盘存储和数据读写速度

尽管数据量和存储容量都在增长，但是磁盘的存取速度，即数据写入磁盘和从其中读出数据的速度并没有同步增长。目前典型的超过平均水平的 1TB 磁盘声称数据访问速率能达到 300Mbps，转速能达到 7200RPM。以此峰值速度读取 1TB 数据需要大约 1 小时（最好的情况下是 55 分钟）。数据量再增加，则花费的时间只增不减。此外，号称转速达到 7200RPM 时访问速率能达到 300Mbps 本身也是误导。传统的旋转介质使用圆形存储磁盘来优化表面积。在圆上，7200RPM 所代表的数据访问量实际取决于所访问的同心圆的圆周。当磁盘逐渐被数据填满以后，周长逐渐变小，导致每次旋转的覆盖面积减少。当磁盘的 65%以上被填满后，300Mbps 的峰值速率会大幅下降。SSD（Solid-state driver，固态硬盘）是旋转介质的一种替代品。SSD 使用微型芯片，而非机电旋转磁盘，它将数据保存在易失性随机访问存储器（volatile random-access memory）中。与旋转介质相比，SSD 承诺更高的速度和 IOPS 性能（input/output operations per second，每秒读写操作次数）。2009 年底到 2010 年初，美光科技（Micron）等公司宣称他们可以提供超过 Gbps 水平访问速度的 SSD（www.dailytech.com/UPDATED+Micron+Announces+Worlds+First+Native+6Gbps+SATA+Solid+State+Drive/article17007.htm）。尽管如此，现状是 SSD 仍然充满 bug 和各种问题，而且成本远高于它的对手磁盘。鉴于数据读写速率受限于磁盘访问速度，因此只能把数据分散到多个存储单元上，而不是集中在单一的大型存储中。

1.1.3 可扩展性

可扩展性是一种能力，有了它系统能通过增加资源提高吞吐量进而解决增加的负荷。可扩展性可以通过两种方式实现，一是配置一个大而强的资源来满足额外的需求，二是依靠由普通机器组成的集群。使用大而强的机器通常属于垂直可扩展性。典型的垂直扩展方案是使用配有大量 CPU 内核且直接挂载大量存储的超级计算机。这类超级计算机通常极其昂贵，属于专有设备。替代垂直扩展的是水平扩展。水平扩展使用商业系统集群，集群随负载的增加而扩展。水平扩展通常需要添加额外的节点来应付额外的负载。

大数据以及大规模并行处理数据的需要促使水平扩展得到了广泛的采纳。在 Google、Amazon、Facebook、eBay 和 Yahoo!，水平扩展的基础设施包含数量巨大的服务器，其中一些包

含几千甚至几十万台服务器。

对水平扩展集群上分布的数据进行处理是非常复杂的事情。在水平集群上处理大规模数据的方法里，MapReduce 模型可能要算是最好的。

1.1.4　MapReduce

MapReduce 这种并行编程模型支持在水平集群上对大规模数据集进行分布式处理。MapReduce 框架是 Google 的专利（http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=/netahtml/PTO/srchnum.htm&r=1&f=G&l=50&s1=7,650,331.PN.&OS=PN/7,650,331&RS=PN/7,650,331），但其核心思想可以自由分享，一些开源实现已经采纳了这些思想。

MapReduce 的创意和灵感来源于函数式编程。map 和 reduce 是函数式编程中两个常用函数。在函数式编程中，map 函数对列表的每个元素执行操作或函数。例如，在列表[1, 2, 3, 4]上执行 multiple-by-two 函数会产生另一个列表[2, 4, 6, 8]。执行这些函数时，原有列表不会被修改。函数式编程认为应当保持数据不可变，避免在多个进程或线程间共享数据。这意味着刚演示过的 map 函数虽然很简单，却可以通过两个或更多线程在同一个列表上同时执行，线程之间互不影响，因为列表本身没有改变。

与 map 函数类似，函数式编程中还有一个 reduce 函数的概念。实际上，reduce 在函数式编程中更广为人知的名字是 fold 函数。reduce 或 fold 函数又称 accumulate、compress 或者 inject 函数。reduce 或 fold 函数对数据结构（例如列表）中的所有元素执行一个函数，最终返回单个结果或输出。因此在 map 函数输出列表[2, 4, 6, 8]上执行 reduce 求和，会得到单个输出值 20。

map 和 reduce 函数可以结合起来处理列表数据，先对列表的每个成员执行一个函数，再对转换生成的列表执行另一个聚合函数。

map 和 reduce 这种简洁的思路可以用在大数据集上，只需稍事修改以适应由元组（tuple）或键/值对组成的集合即可。map 函数对集合中的每组键/值对执行函数并产生一个新集合，接着 reduce 函数对新生成的集合执行聚合以计算最终结果。一个例子胜过千言万语，下面我通过一个简单的例子来解释整个过程。假设存在由键/值对组成的集合：

```
[{ "94303": "Tom"}, {"94303": "Jane"}, {"94301": "Arun"}, {"94302": "Chen"}]
```

其中键是邮政编码，值是邮编指代范围内居民的姓名。假设在集合上执行某 map 函数可以获取特定邮编范围内所有居民的姓名，则此 map 函数输出如下：

```
[{"94303":["Tom", "Jane"]}, {"94301":["Arun"]}, {"94302":["Chen"]}]
```

接着对上面的输出执行某 reduce 函数以计算特定邮编范围内居民的总数，最终输出如下：

```
[{"94303": 2}, {"94301": 1}, {"94302": 1}]
```

用 MapReduce 来进行这么简单的数据处理显得大材小用了，但我的目的是让你理解概念和过程背后的核心思想。

接下来，列举一部分最知名的 NoSQL 产品，然后按照它们的功能和属性进行分类。

1.2　面向列的有序存储

Google Bigtable 的数据存储模型支持面向列，这与 RDBMS 面向行的存储格式截然不同。面向列的存储能高效地存储数据，如果列值不存在就不存储，这样一来遇到 null 值时就能避免浪费空间。

每个数据单元可以看作一组键/值对集合，单元本身通过主标识符（primary identifier）标识，主标识符又叫主键。Bigtable 和其他类似产品喜欢称呼这个主键为行键（row-key）。另外，正如本节标题所描述的那样，单元按顺序排列好进行存储。数据单元按行键排序。要想说清楚面向列的有序存储，还得举个例子。假设有一张表存储与人相关的信息。其中包含以下各列：名字（first_name）、姓氏（last_name）、职业（occupation）、邮编（zip_code）和性别（gender）。表中某人信息如下：

```
first_name: John
last_name: Doe
zip_code: 10001
gender: male
```

同一张表里的另一组数据：

```
first_name: Jane
zip_code: 94303
```

第一条数据的行键为 1，第二条的行键为 2。面向列的有序存储中这两条数据会这样存储：行键 1 的数据会存储在行键 2 的前面，且两条数据相互比邻。

此外，每条数据中只有合法的键/值对才会被存储。在类 Bigtable 的面向列存储中，数据按列族（column-family）存储。列族通常是在配置或启动时定义好，列则不需要预先定义或声明。列可以存储任何数据类型，前提是数据能持久化成 byte 数组。假设上面例子中列族 name 的成员包括列 first_name 和 last_name，列族 location 的成员包括列 zip_code，列族 profile 的成员包括列 gender。

如此一来，上面例子的底层存储由 3 个存储桶组成：name、location 和 profile。每个桶只会保存合法的键/值对，因此列族 name 桶中存储下列值：

```
For row-key: 1
first_name: John
last_name: Doe
For row-key: 2
first_name: Jane
```

列族 location 存储：

```
For row-key: 1
zip_code: 10001
For row-key: 2
zip_code: 94303
```

列族 profile 只有行键 1 对应的数据值，则它存储：

```
For row-key: 1
gender: male
```

在实际存储中，物理上一条数据的列族并不相互隔离，同一行键的所有数据存储在一起。列

族可以看作代表成员列的键，而行键则是代表整条数据的键。

在 Bigtable 和其他类似产品中，数据按顺序连续存储。当数据逐渐填满一个节点后，它会拆分成多个节点。数据不仅在每个节点上是有序的，而且还跨越多节点形成一个更大的有序集。数据持久化方面会有容错的考虑，每份数据都同时维护 3 个副本。大部分类 Bigtable 产品都利用分布式文件系统将数据持久化到磁盘上。分布式文件系统支持将数据存储到集群的多台服务器上。

因为有序，数据按行键查找效率极高。数据访问随机性更小，查找也更简单，就是在序列中查找包含数据的节点。数据插入发生在队列尾部，数据更新则原地进行，不过一般是添加数据的一个新版本到指定单元（cell）里，而非原地覆写（in-place overwrite）。每个单元始终维护多个版本，版本属性通常可配置。

HBase 是以 Google Bigtable 思想为蓝本创建的、广受欢迎的、开源的有序列族存储。有关使用 HBase 存储访问数据的细节在本书多个章节中均有介绍。

HBase 中存储的数据可以通过 MapReduce 进行处理。Hadoop 的 MapReduce 工具可以很容易地用 HBase 作为数据源或数据接收方。

Bigtable 和其他类似产品的技术规范细节包含在下一章的开始部分。Hold 住你的好奇心，或者现在就翻到第 4 章一探究竟。

接下来，我将列举一些类 Bigtable 产品。

要了解和利用 Google 基础设施的思想，最佳方法是从学习 Hadoop（http://hadoop.apache.org）产品族开始。NoSQL 的 Bigtable 存储 HBase 正是 Hadoop 家族中的一员。

下面采用要点句形式列举一些类 Bigtable 开源产品及其属性。

HBase

- **官方在线资源**。http://hbase.apache.org。
- **历史**。2007 年创建于 Powerset（现在属于微软），微软收购 Powerset 前 HBase 被捐赠给了 Apache 基金会。
- **技术和语言**。Java 实现。
- **访问方法**。JRuby 的 shell 支持命令行访问。有 Thrift、Avro、REST 和 protobuf 客户端。对一些编程语言提供支持。发行版本中内置 Java API。

> Protocol Buffer 简称 Protobuf，是 Google 的数据互换格式。有关其更多信息请参阅：http://code.google.com/p/protobuf/。

- **查询语言**。无原生查询语言。由 Hive（http://hive.apache.org）提供类 SQL 接口。
- **开源许可证**。Apache License 版本 2。
- **使用者**。Facebook、StumbleUpon、Hulu、Ning、Mahalo、Yahoo!等。

Thrift 简介

Thrift 是一个软件框架和一种接口定义语言，支持跨语言的服务和 API 开发。用 Thrift 生成的服务能在 C++、Java、Python、PHP、Ruby、Erlang、Perl、Haskell、C#、Cocoa、Smalltalk 和 OCaml 之间无缝地高效互通。Thrift 在 2007 年由 Facebook 创建，它现在是 Apache 孵化器项目。关于它的更多信息请参阅：http://incubator.apache.org/thrift/。

Hypertable

- **官方在线资源**。www.hypertable.org。
- **历史**。2007 年创建于 Zvents，现在是独立开源项目。
- **技术和语言**。C++实现，使用 Google RE2 正则表达式类库。RE2 实现快速高效。Hypertable 承诺性能高于 HBase，有可能节省处理大量数据的时间及成本。
- **访问方法**。支持命令行。支持 Thrift 接口。基于 Thrift 接口支持一系列语言。充满创造力的开发者甚至为 Hypertable 开发了 JDBC 兼容接口。
- **查询语言**。HQL（Hypertable Query Language，Hypertable 查询语言）是一种用于查询 Hypertable 数据的类 SQL 抽象。Hypertable 也有 Hive 适配器。
- **开源许可证**。GNU GPL 版本 2。
- **使用者**。Zvents、百度（中国最大的搜索引擎）和 Rediff（印度最大的门户网站）。

Cloudata

- **官方在线资源**。www.cloudata.org/。
- **历史**。由韩国开发者 YK Kwon 创建（www.readwriteweb.com/hack/2011/02/open-source-bigtable-cloudata.php）。没有太多有关其起源的公开信息。
- **技术和语言**。Java 实现。
- **访问方法**。支持命令行。支持 Thrift、REST 和 Java API。
- **查询语言**。CQL（Cloudata Query Language，Cloudata 查询语言），一种类 SQL 查询语言。
- **开源许可证**。Apache License 版本 2。
- **使用者**。不详。

有序列族存储是非常流行的一类 NoSQL。不过，NoSQL 还包含许多键/值存储和文档数据库。下面介绍键/值存储。

1.3 键/值存储

哈希表（HashMap）或关联数组（associative array）是可以容纳键/值对的最简数据结构。这类数据结构都是万人迷，因为它们极高效，访问数据的时间复杂度为 O(1)。键/值对中的键在集合中是唯一值，容易查找，便于访问数据。

键/值存储各不相同：有的数据保存在内存里，有的能把数据持久化到磁盘里。键/值对可以被分散保存到集群节点中。

Oracle 的 Berkeley DB 键/值存储简单强大，它是纯粹的存储引擎，键和值都是字节数组。其核心存储引擎并不关注键或值的含义，只管保存传入的字节数组对，然后返回同样的数据给调用客户端。Berkeley DB 支持将数据缓存在内存中，随数据增长将其刷入磁盘。它还支持对键索引，帮助更快地查找和访问。Berkeley DB 自 20 世纪 90 年代中期就已存在。在 BSD4.3 迁移到 BSD4.4 的过程中，它被创建出来替换 AT&T 的 NDBM。到了 1996 年，软件公司 Sleepycat 成立，专门负责为 Berkeley DB 提供支持和维护。

另一种常用的键/值存储是缓存。缓存提供应用中使用最多的数据的内存快照。缓存的目的是减少磁盘 I/O。它可以是最简单的映射表，也可以是支持缓存过期策略的健壮系统。作为一种流行策略，缓存广泛应用于计算机软件栈所有层面以提高性能。操作系统、数据库、中间件和各种应用都使用缓存。

像 EHCache（http://ehcache.org/）这样健壮的开源分布式缓存系统广泛应用于各类 Java 应用中，可以将它看作一种 NoSQL 方案。另一种缓存系统 Memcached（http://memcached.org/）在 Web 应用中非常流行，它是开源的高性能对象缓存系统。Memcached 是 2003 年 Brad Fitzpatrick 为 LiveJournal 开发的。除了作为缓存系统，Memcached 还帮助实现有效的内存管理，它会创建一个大虚拟池，然后在节点间按需分配内存。这种方式避免了碎片区域的出现，即某个节点有多余的空闲内存，而其他节点却急缺内存。

随着 NoSQL 运动的势头日益强劲，涌现出一批新的键/值对数据存储。其中一些构建在 Memcached API 上，一些用 Berkeley DB 作为底层存储，另一些则提供全新的替代方案。

这类键/值存储多包含 API，支持 Get-Set 机制以便获取和设置值；少数，如 Redis（http://redis.io/），则提供更丰富的抽象和更强大的 API。Redis 可被视为数据结构服务器，因为除映射表以外它还提供字符串（字符序列）、列表和集合等数据结构，而且它还支持一套丰富的操作来访问不同类型数据结构的数据。

本书涵盖了许多与键/值存储有关的细节。下面列出一些重要产品及其重要属性。针对一些重要特性，这里再次采用要点句的列举方式。

Membase（拟并入 Couchbase， Couchbase 公司成立后开始从 CouchDB 中获得特性）

- **官方在线资源**。www.membase.org/。
- **历史**。NorthScale 公司（后更名为 Membase）于 2009 年启动该项目。从那时起，Zygna 和 NHN 为其作出了重要贡献。Membase 基于 Memcached，支持 Memcached 的文本和二进制协议，它在 Memcached 基础上增加了很多新特性，包括磁盘持久化、数据复制、在线的集群重新配置和数据动态平衡。Membase 的许多核心创建者也是 Memcached 的贡献者。
- **技术和语言**。Erlang、C 和 C++实现。
- **访问方法**。扩展的 Memcached 兼容 API，可替代 Memcached。
- **开源许可证**。Apache License 版本 2。
- **使用者**。Zynga、NHN 等。

Kyoto Cabinet

- **官方在线资源**。http://fallabs.com/kyotocabinet/。

- ❑ **历史**。Kyoto Cabinet 的前身是 Tokyo Cabinet（http://fallabs.com/tokyocabinet/）。Kyoto Cabinet 的数据库就是包含记录的简单数据文件，每条记录是一对键/值，每个键和值都是一组变长二进制数据。
- ❑ **技术和语言**。C++实现。
- ❑ **访问方法**。提供 C、C++、Java、C#、Python、Ruby、Perl、Erlang、OCaml 和 Lua 的 API。由于协议简单，所在存在非常多的客户端。
- ❑ **开源许可证**。GNU GPL 和 GNU LGPL。
- ❑ **使用者**。Mixi 公司。原作者离开 Mixi 加入 Google 前，Mixi 公司赞助了初期大部分工作。博文和邮件列表显示存在大量用户，但没有可参考的公共列表。

Redis

- ❑ **官方在线资源**。http://redis.io/。
- ❑ **历史**。2009 年 Salvatore Sanfilippo 启动该项目。Salvatore 为其初创公司 LLOOGG（http://lloogg.com/）开发了 Redis。虽然目前仍然是独立项目，不过 Redis 的主要作者受雇于 VMware，该公司赞助了 Redis 的开发。
- ❑ **技术和语言**。C 实现。
- ❑ **访问方法**。支持丰富的方法和操作。可以通过 Redis 命令行接口和一系列得到良好维护的客户端类库进行访问，支持语言包括 Java、Python、Ruby、C、C++、Lua、Haskell、AS3 等。
- ❑ **开源许可证**。BSD。
- ❑ **使用者**。Craigslist。

上面罗列的 3 个键/值存储快速灵活，支持存储实时数据、短期内频繁使用的数据，甚至还支持数据完全地持久化。

至今为止列举的键/值存储都为数据存储提供了强一致性模型。然而，在分布式环境中其他一些键/值存储强调可用性高于一致性。它们中大部分灵感来自 Amazon Dynamo，Amamzon Dynamo 也是键/值存储，它承诺卓越的可用行和可扩展性，并成了 Amazon 分布式容错和高可用性系统的骨干。Apache Cassandra、Basho Riak 和 Voldemort 是 Amazon Dynamo 想法的开源实现。

Amazon Dynamo 推出了大量重要的高可用性思想，其中最重要的是最终一致性（eventual consistency）。最终一致性暗示出节点数据更新过程中，副本可能会出现间歇的不一致。最终一致不是不一致，只是与 RDBMS 典型的 ACID（atomicity、consistency、isolation、durabilty，原子性、一致性、隔离性、持久性）一致性相比更弱。

> 本书涵盖了类 Amazon Dynamo 最终一致性数据存储模块的大量细节，但本章不会涉及相关讨论，因为在介绍这些内容前需要先普及一些背景和技术基础。

下面列出 Amazon Dynamo 的相似产品及其部分重要特点。

Cassandra

- **官方在线资源**。http://cassandra.apache.org/。
- **历史**。由 Facebook 开发，2008 年开源。Apache Cassandra 被捐赠给了 Apache 基金会。
- **技术和语言**。Java 实现。
- **访问方法**。支持命令行访问。支持 Thrift 接口和 Java API。存在多种语言的客户端，包括 Java、Python、Grails、PHP、.NET 和 Ruby。支持 Hadoop 集成。
- **查询语言**。查询语言规范正在形成中。
- **开源许可证**。Apache License 版本 2。
- **使用者**。Facebook、Digg、Reddit、Twitter 等。

Voldemort

- **官方在线资源**。http://project-voldemort.com/。
- **历史**。2008 年由 LinkedIn 数据与分析组创建。
- **技术和语言**。Java 实现。可插拔存储引擎，支持 Berkeley DB 和 MySQL。
- **访问方法**。集成 Thrift、Avro 和 protobuf（http://code.google.com/p/protobuf/）接口。可以结合 Hadoop 使用。
- **开源许可证**。Apache License 版本 2。
- **使用者**。LinkedIn。

Riak

- **官方在线资源**。http://wiki.basho.com/。
- **历史**。创建于 Basho，该公司成立于 2008 年。
- **技术和语言**。Erlang 实现。另外还使用了一点点 C 和 JavaScript。
- **访问方法**。支持基于 HTTP 的 JSON 接口和 protobuf 客户端。有 Erlang、Java、Ruby、Python、PHP 和 JavaScript 类库。
- **开源许可证**。Apache License 版本 2。
- **使用者**。Comcast 和 Mochi Media。

Cassandra、Riak 和 Voldemort，所有这三个开源产品都提供了开源 Amazon Dynamo 的能力。Cassandra 和 Riak 的行为和属性分别显现出各自的双重性：Cassandra 同时拥有 Google Bigtable 和 Amazon Dynamo 的属性，而 Riak 既是键/值存储又是文档数据库。

1.4　文档数据库

文档数据库不是文档管理系统。刚接触 NoSQL 的开发者常会混淆文档数据库和文档/内容管理系统。*文档数据库*中的*文档*一词意指文档中松散结构的键/值对集合，通常是 JSON（JavaScript Object Notation，JavaScript 对象表示法），而非一般意义的*文档*或*表格*（尽管它们也能被存储）。

文档数据库把文档当作一个整体，不会将文档分割成多个键/值对。在集合层面上，这使得不同结构的文档可以放在同一个集合里。文档数据库支持文档索引，不仅包括主标识符，还包括文档的属性。当今为数不多的开源文档数据库中，最声名远扬的要数 MongoDB 和 CouchDB。

MongoDB

- **官方在线资源**。www.mongodb.org。
- **历史**。创建于 10gen。
- **技术和语言**。C++实现。
- **访问方法**。支持 JavaScript 命令行接口。支持多语言驱动，包括 C、C#、C++、Erlang、Haskell、Java、JavaScript、Perl、PHP、Python、Ruby 以及 Scala。
- **查询语言**。类 SQL 查询语言。
- **开源许可证**。GNU Affero GPL（http://gnu.org/licenses/agpl-3.0.html）。
- **使用者**。FourSquare、Shutterfly、Intuit、Github 等。

CouchDB

- **官方在线资源**。http://couchdb.apache.org 和 www.couchbase.com。大部分作者属于 Couchbase 公司。
- **历史**。始于 2005 年，2008 年被纳入 Apache 孵化器。
- **技术和语言**。主要用 Erlang 实现，部分 C 实现，JavaScript 执行环境。
- **访问方法**。支持 REST 高于其他一切机制。可以用标准 Web 工具和客户端访问数据库，和访问 Web 资源的方法相同。
- **开源许可证**。Apache License 版本 2。
- **使用者**。Apple、BBC、Canonical、Cern 等，请参阅：http://wiki.apache.org/couchdb/CouchDBin_the_wild。

下一章开始介绍文档数据库的更多细节。

1.5 图形数据库

到此为止已经列出了大部分主流开源NoSQL产品。其他产品，比如图形数据库和XML数据存储，也可以算作NoSQL数据库。本书不会介绍图形和XML数据库，不过在这儿还是列出两种图形数据库：Neo4j和FlockDB，它们可能很有趣，值得了解一下[①]。

Neo4j 是一个兼容 ACID 的图形数据库，便于快速遍历图形。

Neo4j

- **官方在线资源**。http://neo4j.org。
- **历史**。2003 年创建于 Neo Technologies。（没错，这个数据库在 NoSQL 这个词儿流行之前就已经存在了。）
- **技术和语言**。Java 实现。
- **访问方法**。支持命令行接口和 REST 接口。有多种语言客户端，包括 Java、Python、Ruby、Clojure、Scala 和 PHP。

① 最近出现的一个图形数据库 Titan 也引起了一些注意，更多细节请访问：http://thinkaurelius.github.com/titan/。
——译者注

- **查询语言**。支持 SPARQL 协议和 RDF 查询语言。
- **开源许可证**。AGPL。
- **使用者**。Box.net。

FlockDB

- **官方在线资源**。https://github.com/twitter/flockdb。
- **历史**。创建于 Twitter，2010 年开源。最初是为了存储 Twitter 上粉丝的邻接表而设计的。
- **技术和语言**。Scala 实现。
- **访问方法**。支持 Thrift 和 Ruby 客户端。
- **开源许可证**。Apache License 版本 2。
- **使用者**。Twitter。

前面已经介绍了许多 NoSQL 产品。希望现在你已经完成了热身，我们下面要开始进一步学习这些产品，并了解如何有效利用它们。

1.6　小结

本章首先介绍了 NoSQL 的概念、历史和进一步学习所需的基础知识，之后介绍了面向列的有序存储、键/值存储、最终一致性数据库以及文档数据库最基本的知识。此外，本章还给出了一些产品及其核心属性的要点列表。

NoSQL 并不能解决所有问题，而且肯定存在缺点。不过当数据增长到非常大，大到需要分布到许多集群节点上时，大多数产品的扩展性都很好。处理大量数据同样充满挑战，同样需要新的方法。我们知道了 MapReduce 和它的能力，下面章节中将会学习它的使用模式。

当代开发者是在 RDBMS 的陪伴下成长起来的，所以采用 NoSQL 既是采纳新技术，也是改变行为习惯。这也意味着作为开发者，我们需要首先学习 NoSQL，深入理解它，之后才能对其适用性作出自己的判断。此外，NoSQL 中的许多理念非常适合解决大规模可扩展性问题，可用于各种类型的应用。

下一章我们将通过动手实验和概念性介绍来说明面向列存储、键/值存储和文档数据库的结构。我会尽全力提供所有相关信息，但肯定做不到全面覆盖。我不会介绍每个类别的所有产品，只会选择每种类型的代表产品。如果你能将本书从头读到尾，就能在应用程序栈里高效利用 NoSQL。卷起袖子开始学习吧！祝你好运！

第2章

NoSQL 上手初体验

本章内容

- 使用 NoSQL 技术
- 探索 MongoDB 和 Apache Cassandra 基础
- 使用流行高级编程语言访问 MongoDB 和 Apache Cassandra

本章是经典 Hello World 入门的 NoSQL 版，介绍了一些基本的例子。这些例子虽说基础，但并非在控制台上打印 Hello 消息那么简单，而是要让你上手体验 NoSQL。NoSQL 是对一类数据存储的抽象，是一个概念、一种分类，是全新的数据存储观点。它涵盖了一类产品和一系列可以相互替换的非关系型数据存储。在第 1 章我们熟悉了一些基本概念，知道了 NoSQL 的优缺点，从本章开始我们将实战 NoSQL。

本章的例子将用到 MongoDB 和 Cassandra，所以你需要安装设置好它们。如果在开发环境中安装这些产品时需要帮助，请参考附录 A。

为什么只用 MongoDB 和 Apahce Cassandra？

选择使用 MongoDB 和 Cassandra 来演示 NoSQL 例子其实并非刻意而为。本章的目的是提供 NoSQL 这一广阔领域的第一手使用体验。市面上为数众多的 NoSQL 产品大都提供引人注目的功能和优势，要决定选择哪几个产品可不是件简单的事。比方说，可以选择 Couchbase 服务器而非 MongoDB，也可以选择 HBase 而非 Cassandra。样例也可以基于 Redis、Membase、Hypertable 或者 Riak 这样的产品。本书涵盖了许多 NoSQL 数据库，所以只要通读全书，一定会了解 NoSQL 领域的各种替代方案。

2.1　第一印象——两个简单的例子

闲话少说，进入正题，首先介绍两个简单的例子。第一个例子是创建简单的位置偏好数据库，第二个例子是管理汽车品牌和型号数据库。两个例子的重点都在 NoSQL 数据管理。

2.1.1　简单的位置偏好数据集

位置服务现在越来越火了，因为本地企业都在尝试接触周边的用户，大公司则在尝试基于位

置向用户提供个性化在线体验和产品推荐。在一些知名应用中常能见到位置偏好技术的身影，比如 Google Maps 和在线零售商 Walmart.com，前者支持本地搜索，后者根据距离用户最近的沃尔玛超市提供产品和促销信息。

有时用户位置是要求用户输入的，有时是推断出来的。推断可能基于用户 IP 地址、网络接入点（特别是当用户使用移动设备时），或这些技术的各种组合。但是不论如何收集数据，都需要高效地存储这些数据，我们的第一个例子就从这儿展开。

为了简单起见，我们只考虑美国用户的位置偏好，这样只需提供用户标识符和邮编就能找出用户的位置。取用户名作为用户标识符，则要保存的数据类似这样："John Doe, 10001"、"Lee Chang, 94129"、"Jenny Gonzalez 33101"、"Srinivas Shastri, 02101"。

在存储这些数据时，为了保持灵活性和可扩展性，我们选择非关系型数据库产品 MongoDB。接下来创建 MongoDB 数据库，存储样例位置数据。

1. 启动 MongoDB 和存储数据

假定你已经成功地安装好 MongoDB，现在要启动服务器，然后连上它。

通过运行发行版 `bin` 目录下的 `mongod` 程序可以启动 MongoDB 服务器。针对不同的环境，比如 Windows、Mac OS X，或者某版 Linux，MongoDB 的版本会有所不同，但是所有版本中服务器程序的名字都一样，而且都在 bin 目录下。

连接 MongoDB 服务器最简单的办法就是用它自带的 JavaScript Shell，在命令行界面运行 `mongo` 即可。`Mongo` 的 JavaScript Shell 命令也在 `bin` 目录下。

执行 `mongod` 启动 MongoDB 服务器后，会在控制台里看到类似如下的输出：

```
PS C:\applications\mongodb-win32-x86_64-1.8.1> .\bin\mongod.exe
C:\applications\mongodb-win32-x86_64-1.8.1\bin\mongod.exe
--help for help and startup options
Sun May 01 21:22:56 [initandlisten] MongoDB starting : pid=3300 port=27017
  dbpath=/data/db/ 64-bit
Sun May 01 21:22:56 [initandlisten] db version v1.8.1, pdfile version 4.5
Sun May 01 21:22:56 [initandlisten] git version:
a429cd4f535b2499cc4130b06ff7c26f41c00f04
Sun May 01 21:22:56 [initandlisten] build sys info: windows (6, 1, 7600, 2, ''
  BOOST_LIB_VERSION=1_42
Sun May 01 21:22:56 [initandlisten] waiting for connections on port 27017
Sun May 01 21:22:56 [websvr] web admin interface listening on port 28017
```

上面的输出来自 Windows 7 64 位，Windows PowerShell 环境。受环境影响你的输出可能会有所变化。

现在，数据库服务器已经启动运行起来，用 `mongo` JavaScript shell 连上它。Shell 最开始的输出应该是这样的：

```
PS C:\applications\mongodb-win32-x86_64-1.8.1> bin/mongo
MongoDB shell version: 1.8.1
connecting to: test
>
```

`mongo` shell 默认连接本地的"test"数据库。从 `mongod`（服务器守护程序）的控制台输出中，可以猜到 MongoDB 服务器在端口 27017 上等待连接。要了解初始可用的命令，只需输入 `help`

按下回车（或返回），就会看到下面这样的命令选项列表：

```
> help
        db.help()                    help on db methods
        db.mycoll.help()             help on collection methods
        rs.help()                    help on replica set methods
        help connect                 connecting to a db help
        help admin                   administrative help
        help misc                    misc things to know
        help mr                      mapreduce help

        show dbs                     show database names
        show collections             show collections in current database
        show users                   show users in current database
        show profile                 show most recent system.profile entries
                                         with time >= 1ms
        use <db_name>                set current database
        db.foo.find()                list objects in collection foo
        db.foo.find( { a : 1 } )     list objects in foo where a == 1
        it                           result of the last line evaluated;
                                         use to further iterate
        DBQuery.shellBatchSize = x   set default number of items to display
                                         on shell
        exit                         quit the mongo shell
>
```

自定义 MongoDB 数据目录和端口号

MongoDB 默认在目录/data/db（Windows 上是 C:\data\db）下存储数据文件，在端口 27017 上监听请求。使用 dbpath 选项可以指定一个不同的数据目录：

```
mongod --dbpath  /path/to/alternative/directory
```

如果数据目录不存在，要先创建好，还要确保 mongod 拥有该目录的写权限。

此外，使用 port 选项可以指定不同的连接端口号：

```
mongod --port 94301
```

为避免冲突，应首先确认该端口号无人使用。

要同时修改数据目录和端口号，同时指定--dbpath 和--port 选项即可。

下面学习如何在 MongoDB 中创建偏好数据库。

2. 创建偏好数据库

第一步先创建偏好数据库 prefs，然后是库中名为 location 的集合，最后在该集合中按用户名加邮编的元组（或对）结构存储数据。按照 MongoDB 的说法就是执行下列步骤。

(1) 切换到 prefs 数据库。

(2) 定义要存储的数据集。

(3) 将定义好的数据集保存到集合中，集合名为 location。

要执行这些步骤，在 Mongo JavaScript 控制台中键入：

```
use prefs
w = {name: "John Doe", zip: 10001};
x = {name: "Lee Chang", zip: 94129};
y = {name: "Jenny Gonzalez", zip: 33101};
z = {name: "Srinivas Shastri", zip: 02101};
db.location.save(w);
db.location.save(x);
db.location.save(y);
db.location.save(z);
```

搞定！只需简单几步，数据存储就准备妥当了。提醒几点，`use prefs` 命令将当前数据库改变为 `prefs` 数据库，但是数据库本身一直都没有显式地创建过。同样，通过传递每条数据给 `db.location.save()`方法将数据存入 `location` 集合，但是集合也没有显式地创建过。在 MongoDB 中，数据库和集合都是在插入数据时才创建。因此，在这个例子里，创建是发生在插入第一条数据`{name: "John Doe", zip: 10001}`时。

下面查询新建数据库来验证存储内容。要获得 `location` 集合中的所有记录，可以执行 `db.location.find()`。

在我的机器上执行 `db.location.find()`会显示下列输出：

```
> db.location.find()
{ "_id" : ObjectId("4c97053abe67000000003857"), "name" : "John Doe",
    "zip" : 10001 }
{ "_id" : ObjectId("4c970541be67000000003858"), "name" : "Lee Chang",
    "zip" : 94129 }
{ "_id" : ObjectId("4c970548be67000000003859"), "name" : "Jenny Gonzalez",
    "zip" : 33101 }
{ "_id" : ObjectId("4c970555be6700000000385a"), "name" : "Srinivas Shastri",
    "zip" : 1089 }
```

你的输出应该非常接近，唯一可能不同的地方是 `ObjectId.ObjectId`，在 MongoDB 中它用来唯一标识每条记录或每个文档。

> MongoDB 使用 `ObjectId` 来唯一标识集合中的每个文档。文档的 `ObjectId` 存储在其`_id` 属性里。插入记录时，任何唯一值都可以成为 `ObjectId`，其唯一性由开发者保障。插入记录时也可以不提供`_id` 属性值。这种情况下，MongoDB 会创建并插入一个适当的唯一的标识符。MongoDB 生成的标识符是 BSON 格式，即二进制 JSON 格式，格式概述如下。
>
> - BSON ObjectId 是一个 12 字节的值。
> - 前 4 字节是创建时间，表示自纪元以来的秒数。这个值必须用大端（Big endian）模式存储，即最高数位的值存储在最低的存储地址上。
> - 接下来 3 字节表示机器标识符。
> - 跟着 2 字节表示进程标识符。
> - 最后 3 字节表示计数器，这个值也必须以大端存储。
> - BSON 格式除了保证唯一性以外，还包含了创建时间。所有标准的 MongoDB 驱动都支持 BSON 格式标识符。

调用 find 方法如果不带参数则返回集合中的所有元素，但是某些情况下可能只需要集合的部分子集。为便于了解都能执行哪些查询，先在 location 集合中加入下列记录：

- Don Joe, 10001
- John Doe, 94129

可通过 mongo shell 完成插入，如下所示：

```
> a = {name:"Don Joe", zip:10001};
{ "name" : "Don Joe", "zip" : 10001 }
> b = {name:"John Doe", zip:94129};
{ "name" : "John Doe", "zip" : 94129 }
> db.location.save(a);
> db.location.save(b);
>
```

要获得邮编 10001 的所有用户名列表，可以这样查询：

```
> db.location.find({zip: 10001});
{ "_id" : ObjectId("4c97053abe67000000003857"), "name" : "John Doe",
   "zip" : 10001 }
{ "_id" : ObjectId("4c97a6555c760000000054d8"), "name" : "Don Joe",
   "zip" : 10001 }
```

要获得名字是“John Doe”的所有记录列表，可以这样查询：

```
> db.location.find({name: "John Doe"});
{ "_id" : ObjectId("4c97053abe67000000003857"), "name" : "John Doe",
   "zip" : 10001 }
{ "_id" : ObjectId("4c97a7ef5c760000000054da"), "name" : "John Doe",
   "zip" : 94129 }
```

在这两个过滤集合的查询中，查询文档作为参数被传递给 find 方法，其中指定了键和值要匹配的模式。除了这些简单的过滤条件外，MongoDB 还支持许多高级查询机制，包括借助了正则表达式的模式。

只要是数据库就总会迎来新数据，所以集合结构就有可能逐渐变成约束，进而需要修改。在传统的关系型数据库中，就可能需要修改表结构。修改表结构意味着执行复杂的数据迁移任务以确保新老结构的数据可以共存。在 MongoDB 中，修改集合结构是小菜一碟。更准确地说，集合（类似于表）是弱结构的，所以在同一个集合中允许存储不同的文档类型。

举个例子，假设现在又要存储另一个用户的位置偏好，其姓名和邮编与数据库中一个已经存在的文档完全相同，比如说{name:"Lee Chang", zip: 94129}。当然，这是故意的，我们假设姓名和邮编对应该唯一。

为了区分第二个 Lee Chang 和数据库中已存在的那个，可以添加一个额外属性，即街道地址：

```
> anotherLee = {name:"Lee Chang", zip: 94129, streetAddress:"37000 Graham Street"};
{
        "name" : "Lee Chang",
        "zip" : 94129,
        "streetAddress" : "37000 Graham Street"
}
> db.location.save(anotherLee);
```

现在再用 `find` 获取所有文档，返回如下数据集：

```
> db.location.find();
{ "_id" : ObjectId("4c97053abe67000000003857"), "name" : "John Doe",
    "zip" : 10001 }
{ "_id" : ObjectId("4c970541be67000000003858"), "name" : "Lee Chang",
   "zip" : 94129 }
{ "_id" : ObjectId("4c970548be67000000003859"), "name" : "Jenny Gonzalez",
    "zip" : 33101 }
{ "_id" : ObjectId("4c970555be6700000000385a"), "name" : "Srinivas Shastri",
    "zip" : 1089 }
{ "_id" : ObjectId("4c97a6555c760000000054d8"), "name" : "Don Joe",
    "zip" : 10001 }
{ "_id" : ObjectId("4c97a7ef5c760000000054da"), "name" : "John Doe",
    "zip" : 94129 }
{ "_id" : ObjectId("4c97add25c760000000054db"), "name" : "Lee Chang",
    "zip" : 94129, "streetAddress" : "37000 Graham Street" }
```

通过各种主流编程语言可以访问数据，因为各种驱动都存在。本章稍后的 2.2 节涵盖了有关话题。在 2.2 节的部分内容中，将使用 Java、PHP、Ruby 和 Python 访问位置偏好数据。

在接下来的例子里，将使用非关系型列族数据库存储汽车品牌和型号数据。

2.1.2　存储汽车品牌和型号数据

本例中使用分布式列族数据库 Apache Cassandra。因此，在继续深入研究本例之前应先安装好 Cassandra。如果在安装和设置 Cassandra 时需要帮助，请参阅附录 A。

Apache Cassandra 是分布式数据库，所以使用时通常会建立数据库集群。在这个例子中，设置 Cassandra 作为单个节点运行能避免集群设置的复杂性。生产环境则不会这样配置，但对投石问路和了解基础知识来说，单个节点足矣。

Cassandra 数据库可以通过单个命令行客户端或者 Thrift 接口访问。Thrift 接口支持多种编程语言。从功能角度讲，可以把 Thrift 接口当作通用的多语言数据库驱动。关于 Thrift 的内容会在稍后的 2.2 节继续讨论。

要继续学习汽车品牌和型号数据库，首先应启动 Cassandra 并连接它。

1. 启动与连接 Cassandra

Cassandra 发行版压缩包（tar 打包，gzip 压缩）解压后的目录里有个文件 `bin/cassandra`，调用这个文件可以启动 Cassandra 服务器。运行 `bin/Cassandra-f` 启动一个本地 Cassandra 节点，参数 `-f` 让 Cassandra 在前台运行。如果运行 Cassandra 集群，则会启动多节点，而且节点间可以相互通信。在这个例子中，演示如何使用 Cassandra 存储和访问数据，用一个节点就够了。

启动 Cassandra 节点后，应该会看到如下输出：

```
PS C:\applications\apache-cassandra-0.7.4> .\bin\cassandra -f
Starting Cassandra Server
 INFO 18:20:02,091 Logging initialized
 INFO 18:20:02,107 Heap size: 1070399488/1070399488
 INFO 18:20:02,107 JNA not found. Native methods will be disabled.
 INFO 18:20:02,107 Loading settings from file:/C:/applications/
    apache-cassandra-0.7.4/conf/cassandra.yaml
 INFO 18:20:02,200 DiskAccessMode 'auto' determined to be standard,
    indexAccessMode is standard
 INFO 18:20:02,294 Deleted \var\lib\cassandra\data\system\LocationInfo-f-3
 INFO 18:20:02,294 Deleted \var\lib\cassandra\data\system\LocationInfo-f-2
 INFO 18:20:02,294 Deleted \var\lib\cassandra\data\system\LocationInfo-f-1
 INFO 18:20:02,310 Deleted \var\lib\cassandra\data\system\LocationInfo-f-4
 INFO 18:20:02,341 Opening \var\lib\cassandra\data\system\LocationInfo-f-5
 INFO 18:20:02,388 Couldn't detect any schema definitions in local storage.
 INFO 18:20:02,388 Found table data in data directories. Consider using JMX to call
    org.apache.cassandra.service.StorageService.loadSchemaFromYam
l().
 INFO 18:20:02,403 Creating new commitlog segment /var/lib/cassandra/commitlog\
    CommitLog-1301793602403.log
 INFO 18:20:02,403 Replaying \var\lib\cassandra\commitlog\
    CommitLog-1301793576882.log
 INFO 18:20:02,403 Finished reading \var\lib\cassandra\commitlog\
    CommitLog-1301793576882.log
 INFO 18:20:02,419 Log replay complete
 INFO 18:20:02,434 Cassandra version: 0.7.4
 INFO 18:20:02,434 Thrift API version: 19.4.0
 INFO 18:20:02,434 Loading persisted ring state
 INFO 18:20:02,434 Starting up server gossip
 INFO 18:20:02,450 Enqueuing flush of Memtable-LocationInfo@33000296(29 bytes,
    1 operations)
 INFO 18:20:02,450 Writing Memtable-LocationInfo@33000296(29 bytes, 1 operations)
 INFO 18:20:02,622 Completed flushing \var\lib\cassandra\data\system\
    LocationInfo-f-6-Data.db (80 bytes)
 INFO 18:20:02,653 Using saved token 63595432991552520182800882743159853717
 INFO 18:20:02,653 Enqueuing flush of Memtable-LocationInfo@22518320(53 bytes,
    2 operations)
 INFO 18:20:02,653 Writing Memtable-LocationInfo@22518320(53 bytes, 2 operations)
 INFO 18:20:02,824 Completed flushing \var\lib\cassandra\data\system\
    LocationInfo-f-7-Data.db (163 bytes)
 INFO 18:20:02,824 Will not load MX4J, mx4j-tools.jar is not in the classpath
 INFO 18:20:02,871 Binding thrift service to localhost/127.0.0.1:9160
 INFO 18:20:02,871 Using TFastFramedTransport with a max frame size of
    15728640 bytes.
 INFO 18:20:02,871 Listening for thrift clients...
```

上面的输出是从我的 Windows 7 64 位，Windows PowerShell 中得到的。如果使用不同的操作系统和 shell，输出可能会有所不同。

Apache Cassandra 节点基本配置

Apache Cassandra 存储配置定义在文件 conf/Cassandra.yaml 中。Cassandra 稳定版或开发版的 tar.gz 格式压缩包中的 cassandra.yaml 文件包含一些默认配置。例如，提交日志放在 /var/lib/Cassandra/commitlog 目录中，数据文件放在/var/lib/Cassandra/data 目录中。Apache Cassandra 使用 log4j 记录日志。Cassandra log4j 通过文件 conf/log4j-server.properties 配置。默认情况下，Cassandra log4j 将日志输出写到/var/log/Cassandra/system.log 中。如果想保留默认配置，请确保以上目录存在并且具备读写权限。如果想修改配置，请确保在对应日志文件中指定新文件夹。

我的提交日志和数据目录属性分别是：

```
# directories where Cassandra should store data on disk.data_file_directories:
    - /var/lib/cassandra/data

# commit log
    commitlog_directory: /var/lib/cassandra/commitlog
```

cassandra.yaml 中的路径不必是 Windows 格式。例如，不需要指定提交日志路径为 commitlog_directory:C:\var\lib\cassandra\commitlog。我的 conf/log4j-server.properties 里 log4j appender 的文件配置如下：

```
log4j.appender.R.File=/var/log/cassandra/system.log
```

连接运行中的 Cassandra 节点的最简单办法是使用 Cassandra CLI（Command-Line Interface，命令行接口）。只需要运行 bin/Cassandra-cli 就可以启动命令行，可以指定主机地址和端口号如下：

```
bin/cassandra-cli -host localhost -port 9160
```

运行 cassandra-cli 输出如下：

```
PS C:\applications\apache-cassandra-0.7.4> .\bin\cassandra-cli -host localhost
    -port 9160
Starting Cassandra Client
Connected to: "Test Cluster" on localhost/9160
Welcome to cassandra CLI.

Type 'help;' or '?' for help. Type 'quit;' or 'exit;' to quit.
[default@unknown]
```

输入 help 或?得到命令列表：

```
[default@unknown] ?
List of all CLI commands:
?                                                           Display this message.
help;                                                          Display this help.
help <command>;                               Display detailed, command-specific help.
connect <hostname>/<port> (<username> '<password>')?; Connect to thrift service.
use <keyspace> [<username> 'password'];                     Switch to a keyspace.
describe keyspace (<keyspacename>)?;                           Describe keyspace.
exit;                                                                   Exit CLI.
quit;                                                                   Exit CLI.
describe cluster;                                 Display information about cluster.
```

```
show cluster name;                                        Display cluster name.
show keyspaces;                                           Show list of keyspaces.
show api version;                                         Show server API version.
create keyspace <keyspace> [with <att1>=<value1> [and <att2>=<value2> ...]];
                Add a new keyspace with the specified attribute(s) and value(s).
update keyspace <keyspace> [with <att1>=<value1> [and <att2>=<value2> ...]];
                 Update a keyspace with the specified attribute(s) and value(s).
create column family <cf> [with <att1>=<value1> [and <att2>=<value2> ...]];
        Create a new column family with the specified attribute(s) and value(s).
update column family <cf> [with <att1>=<value1> [and <att2>=<value2> ...]];
            Update a column family with the specified attribute(s) and value(s).
drop keyspace <keyspace>;                                      Delete a keyspace.
drop column family <cf>;                                  Delete a column family.
get <cf>['<key>'];                                        Get a slice of columns.
get <cf>['<key>']['<super>'];                         Get a slice of sub columns.
get <cf> where <column> = <value> [and <column> > <value> and ...] [limit int];
get <cf>['<key>']['<col>'] (as <type>)*;                      Get a column value.
get <cf>['<key>']['<super>']['<col>'] (as <type>)*;       Get a sub column value.
set <cf>['<key>']['<col>'] = <value> (with ttl = <secs>)*;          Set a column.
set <cf>['<key>']['<super>']['<col>'] = <value> (with ttl = <secs>)*;
                                                                Set a sub column.
del <cf>['<key>'];                                                 Delete record.
del <cf>['<key>']['<col>'];                                        Delete column.
del <cf>['<key>']['<super>']['<col>'];                         Delete sub column.
count <cf>['<key>'];                                     Count columns in record.
count <cf>['<key>']['<super>'];                  Count columns in a super column.
truncate <column_family>;                         Truncate specified column family.
assume <column_family> <attribute> as <type>;
              Assume a given column family attributes to match a specified type.
list <cf>;                                      List all rows in the column family.
list <cf>[<startKey>:];
                       List rows in the column family beginning with <startKey>.
list <cf>[<startKey>:<endKey>];
        List rows in the column family in the range from <startKey> to <endKey>.
list ... limit N;                                      Limit the list results to N.
```

熟悉了 Cassandra 的基础知识之后，下面我们为汽车品牌和型号数据创建存储定义，然后使用新定义好的 Cassandra 存储结构插入和读取数据。

2. 使用 Cassandra 存储和访问数据

首先要了解键空间和列族的概念。关系型数据库中最近似键空间和列族的概念是数据库和表。虽然这些定义不完全准确，而且有时可能引起误解，但是它们有助于理解如何使用键空间和列族。在熟悉过基本使用模式以后，就会更好地理解这些概念，它们已经超越了对等的关系型概念。

下面先列出 Cassandra 服务器中已经存在的键空间。打开 cassandra-cli，键入 `show keyspaces` 命令，然后按回车。因为使用了全新安装的 Cassandra，输出类似这样：

```
[default@unknown] show keyspaces;
Keyspace: system:
  Replication Strategy: org.apache.cassandra.locator.LocalStrategy
    Replication Factor: 1
  Column Families:
    ColumnFamily: HintsColumnFamily (Super)
    "hinted handoff data"
```

```
        Columns sorted by: org.apache.cassandra.db.marshal.BytesType/
      org.apache.cassandra.db.marshal.BytesType
        Row cache size / save period: 0.0/0
        Key cache size / save period: 0.01/14400
        Memtable thresholds: 0.15/32/1440
        GC grace seconds: 0
        Compaction min/max thresholds: 4/32
        Read repair chance: 0.0
        Built indexes: []
      ColumnFamily: IndexInfo
      "indexes that have been completed"
        Columns sorted by: org.apache.cassandra.db.marshal.UTF8Type
        Row cache size / save period: 0.0/0
        Key cache size / save period: 0.01/14400
        Memtable thresholds: 0.0375/8/1440
        GC grace seconds: 0
        Compaction min/max thresholds: 4/32
        Read repair chance: 0.0
        Built indexes: []
      ColumnFamily: LocationInfo
      "persistent metadata for the local node"
        Columns sorted by: org.apache.cassandra.db.marshal.BytesType
        Row cache size / save period: 0.0/0
        Key cache size / save period: 0.01/14400
        Memtable thresholds: 0.0375/8/1440
        GC grace seconds: 0
        Compaction min/max thresholds: 4/32
        Read repair chance: 0.0
        Built indexes: []
      ColumnFamily: Migrations
      "individual schema mutations"
        Columns sorted by: org.apache.cassandra.db.marshal.TimeUUIDType
        Row cache size / save period: 0.0/0
    Key cache size / save period: 0.01/14400
    Memtable thresholds: 0.0375/8/1440
    GC grace seconds: 0
    Compaction min/max thresholds: 4/32
    Read repair chance: 0.0
    Built indexes: []
  ColumnFamily: Schema
  "current state of the schema"
    Columns sorted by: org.apache.cassandra.db.marshal.UTF8Type
    Row cache size / save period: 0.0/0
    Key cache size / save period: 0.01/14400
    Memtable thresholds: 0.0375/8/1440
    GC grace seconds: 0
    Compaction min/max thresholds: 4/32
    Read repair chance: 0.0
    Built indexes: []
```

系统键空间，顾名思义，就像 RDBMS 里的管理数据库。系统键空间包括了一些预定义好的列族，后面小节中会通过例子介绍列族的知识。键空间将列族组织到一起。通常来说，每个应用程序里定义一个键空间。数据复制定义在键空间层面上，即在键空间一级声明数据副本的数量和存储方式。Cassandra 发行版中自带一个示例键空间创建脚本，在 conf 目录下的 schema-sample.txt

文件中。运行示例键空间创建脚本如下：

```
PS C:\applications\apache-cassandra-0.7.4> .\bin\cassandra-cli -host localhost
  --file .\conf\schema-sample.txt
```


我们再次通过命令行客户端连接，在命令行中重发 show keyspaces 命令，这次输出应该会像这样：

```
[default@unknown] show keyspaces;
Keyspace: Keyspace1:
  Replication Strategy: org.apache.cassandra.locator.SimpleStrategy
    Replication Factor: 1
  Column Families:
    ColumnFamily: Indexed1
      Columns sorted by: org.apache.cassandra.db.marshal.BytesType
      Row cache size / save period: 0.0/0
      Key cache size / save period: 200000.0/14400
      Memtable thresholds: 0.2953125/63/1440
      GC grace seconds: 864000
      Compaction min/max thresholds: 4/32
      Read repair chance: 1.0
      Built indexes: [Indexed1.birthdate_idx]
      Column Metadata:
        Column Name: birthdate (626972746864617465)
          Validation Class: org.apache.cassandra.db.marshal.LongType
          Index Name: birthdate_idx
          Index Type: KEYS
    ColumnFamily: Standard1
      Columns sorted by: org.apache.cassandra.db.marshal.BytesType
  Row cache size / save period: 1000.0/0
  Key cache size / save period: 10000.0/3600
  Memtable thresholds: 0.29/255/59
  GC grace seconds: 864000
  Compaction min/max thresholds: 4/32
  Read repair chance: 1.0
  Built indexes: []
ColumnFamily: Standard2
  Columns sorted by: org.apache.cassandra.db.marshal.UTF8Type
  Row cache size / save period: 0.0/0
  Key cache size / save period: 100.0/14400
  Memtable thresholds: 0.2953125/63/1440
  GC grace seconds: 0
  Compaction min/max thresholds: 5/31
  Read repair chance: 0.0010
  Built indexes: []
ColumnFamily: StandardByUUID1
  Columns sorted by: org.apache.cassandra.db.marshal.TimeUUIDType
  Row cache size / save period: 0.0/0
  Key cache size / save period: 200000.0/14400
  Memtable thresholds: 0.2953125/63/1440
  GC grace seconds: 864000
  Compaction min/max thresholds: 4/32
  Read repair chance: 1.0
  Built indexes: []
```

```
ColumnFamily: Super1 (Super)
  Columns sorted by: org.apache.cassandra.db.marshal.BytesType/
org.apache.cassandra.db.marshal.BytesType
  Row cache size / save period: 0.0/0
  Key cache size / save period: 200000.0/14400
  Memtable thresholds: 0.2953125/63/1440
  GC grace seconds: 864000
  Compaction min/max thresholds: 4/32
  Read repair chance: 1.0
  Built indexes: []
ColumnFamily: Super2 (Super)
"A column family with supercolumns, whose column and subcolumn names are
UTF8 strings"
  Columns sorted by: org.apache.cassandra.db.marshal.BytesType/
org.apache.cassandra.db.marshal.UTF8Type
  Row cache size / save period: 10000.0/0
  Key cache size / save period: 50.0/14400
  Memtable thresholds: 0.2953125/63/1440
  GC grace seconds: 864000
  Compaction min/max thresholds: 4/32
  Read repair chance: 1.0
  Built indexes: []
ColumnFamily: Super3 (Super)
"A column family with supercolumns, whose column names are Longs (8 bytes)"
  Columns sorted by: org.apache.cassandra.db.marshal.LongType/
org.apache.cassandra.db.marshal.BytesType
  Row cache size / save period: 0.0/0
  Key cache size / save period: 200000.0/14400
  Memtable thresholds: 0.2953125/63/1440
  GC grace seconds: 864000
      Compaction min/max thresholds: 4/32
      Read repair chance: 1.0
      Built indexes: []
Keyspace: system:
...(Information on the system keyspace is not included here as it's
    the same as what you have seen earlier in this section)
```

接下来，创建 CarDataStore 键空间，然后使用代码清单 2-1 中的脚本在其中创建 Cars 列族。

代码清单 2-1　CarDataStore 键空间的定义脚本

```
/*schema-cardatastore.txt*/

create keyspace CarDataStore
    with replication_factor = 1
    and placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy';

use CarDataStore;

create column family Cars
```

```
    with comparator = UTF8Type
    and read_repair_chance = 0.1
    and keys_cached = 100
    and gc_grace = 0
    and min_compaction_threshold = 5
    and max_compaction_threshold = 31;
```

schema-cardatastore.txt

执行代码清单 2-1 的脚本如下：

```
PS C:\applications\apache-cassandra-0.7.4> bin/cassandra-cli -host localhost
    --file C:\workspace\nosql\examples\schema-cardatastore.txt
```

添加键空间成功！现在简要回顾一下如何添加键空间。首先添加名为 `CarDataStore` 的键空间，然后在此键空间中添加一个 `ColumnFamily`，名字叫 Cars。过会儿还会看到 `ColumnFamily`，现在暂且把它们看作表。在 `ColumnFamily` 中还包括一个叫做 `CompareWith` 的属性，值为 `UTF8Type`，它将影响行键的索引和排序方式。键空间里的其他部分用来声明复制选项。`CarDataStore` 的复制因子值为 1，即 Cassandra 中只存储一个数据副本。

接下来，添加一些数据到 `CarDataStore` 键空间中，如下所示：

```
 [default@unknown] use CarDataStore;
Authenticated to keyspace: CarDataStore
[default@CarDataStore] set Cars['Prius']['make'] = 'toyota';
Value inserted.
[default@CarDataStore] set Cars['Prius']['model'] = 'prius 3';
Value inserted.
[default@CarDataStore] set Cars['Corolla']['make'] = 'toyota';
Value inserted.
[default@CarDataStore] set Cars['Corolla']['model'] = 'le';
Value inserted.
[default@CarDataStore] set Cars['fit']['make'] = 'honda';
Value inserted.
[default@CarDataStore] set Cars['fit']['model'] = 'fit sport';
Value inserted.
[default@CarDataStore] set Cars['focus']['make'] = 'ford';
Value inserted.
[default@CarDataStore] set Cars['focus']['model'] = 'sel';
Value inserted.
```

上面演示的是在 Cassandra 中使用命令添加数据。使用此命令可以添加一个名–值对或者列值到一行里，它们定义在键空间的 `ColumnFamily` 中。比如命令 `set Cars['Prius']['make'] = 'toyota'`添加了名–值对 `'make' = 'toyota'`，键为`'Prius'`。`'Prius'`所标识的行是 `Cars ColumnFamily` 的一部分，而 `Cars ColumnFamily` 则定义在键空间 `CarDataStore` 中。

添加完数据后，可以查询和访问数据。要获取 `Prius` 标识的名–值对（列名和列值），使用命令：`get Cars['Prius']`。输出如下所示：

```
[default@CarDataStore] get Cars['Prius'];
=> (column=make, value=746f796f7461, timestamp=1301824068109000)
=> (column=model, value=70726975732033, timestamp=1301824129807000)
Returned 2 results.
```

查询时要注意行键、列族标识符和列键都是区分大小写的。因此使用'prius'（而不是'Prius'）不会返回任何名-值对。试试运行 get Cars['prius']，应该会得到响应 Returned 0 results。此外注意，查询前要记得发送命令 use CarDataStore，让 CarDataStore 成为当前的键空间。

如果只想得到'Prius'行的'make'名–值对，可以这样查询：

```
[default@CarDataStore] get Cars['Prius']['make'];
=> (column=make, value=746f796f7461, timestamp=1301824068109000)
```

Cassandra 支持的数据模型远比这里展示的丰富，查询能力也比这里演示的更复杂，不过这些话题还是留到后面章节再谈。我相信现在你已经有点感觉了。

前面介绍了两个简单的例子，一个使用文档存储 MongoDB，一个使用列族数据库 Apache Cassandra，下面我们使用编程语言来访问它们。

2.2 使用多种语言

要把 NoSQL 嵌入应用程序栈里，极其重要的一点是强健而灵活的多语言应用，即能使用最流行的语言访问和操纵这些存储。

本节介绍 NoSQL 存储和编程语言之间的两种接口。首先是 Java、PHP、Ruby 和 Python 版本的 MongoDB 驱动，其次是 Apache Cassandra 的与语言无关（支持多语言）的 Thrift 接口。这些主题涵盖的都是非常基本的内容，后面的章节会在这个基础上展示更强大、更详细的用例。

2.2.1 MongoDB驱动

本节介绍四种不同语言的 MongoDB 驱动，依次是 Java、PHP、Ruby 和 Python。

1. MongoDB 的 Java 驱动

首先从 MongoDB 的 github 代码库（http://github.com/mongodb）中下载最新版 MongoDB Java 驱动。所有官方支持的驱动都托管在这个代码库中。最新版本的驱动是 2.5.2，所以下载名为 mongo-2.5.2,jar 的 jar 文件。

我们再次运行 bin/mongod 启动本地 MongoDB 服务器。这次用 Java 程序来连接服务器。请查看代码清单 2-2 中的 Java 样例程序，它连接 MongoDB，列出 prefs 数据库的所有集合，然后列出 location 集合中的所有文档。

代码清单 2-2 连接 MongoDB 的 Java 样例程序

```
import java.net.UnknownHostException;
import java.util.Set;
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.Mongo;
import com.mongodb.MongoException;
```

```
public class ConnectToMongoDB {
    Mongo m = null;
    DB db;

    public void connect() {
        try {
            m = new Mongo("localhost", 27017 );
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (MongoException e) {
            e.printStackTrace();
        }
    }

    public void listAllCollections(String dbName) {
        if(m!=null){
            db = m.getDB(dbName);
            Set<String> collections = db.getCollectionNames();

            for (String s : collections) {
                System.out.println(s);
            }
        }
    }

    public void listLocationCollectionDocuments() {
        if(m!=null){
            db = m.getDB("prefs");
            DBCollection collection = db.getCollection("location");

            DBCursor cur = collection.find();

            while(cur.hasNext()) {
                System.out.println(cur.next());
            }
        } else {
            System.out.println("Please connect to MongoDB
    and then fetch the collection");
        }
    }

    public static void main(String[] args) {
        ConnectToMongoDB connectToMongoDB = new ConnectToMongoDB();
        connectToMongoDB.connect();
        connectToMongoDB.listAllCollections("prefs");
        connectToMongoDB.listLocationCollectionDocuments();
    }
}
```

ConnectToMongoDB.java

编译和运行该程序时，请确保 MongoDB Java 驱动在 classpath 里。运行程序输出如下：

```
location
system.indexes
{ "_id" : { "$oid" : "4c97053abe67000000003857"} , "name" : "John Doe" ,
    "zip" : 10001.0}
{ "_id" : { "$oid" : "4c970541be67000000003858"} , "name" : "Lee Chang" ,
    "zip" : 94129.0}
{ "_id" : { "$oid" : "4c970548be67000000003859"} , "name" : "Jenny Gonzalez" ,
    "zip" : 33101.0}
{ "_id" : { "$oid" : "4c970555be6700000000385a"} , "name" : "Srinivas Shastri" ,
    "zip" : 1089.0}
{ "_id" : { "$oid" : "4c97a6555c760000000054d8"} , "name" : "Don Joe" ,
    "zip" : 10001.0}
{ "_id" : { "$oid" : "4c97a7ef5c760000000054da"} , "name" : "John Doe" ,
    "zip" : 94129.0}
{ "_id" : { "$oid" : "4c97add25c760000000054db"} , "name" : "Lee Chang" ,
    "zip" : 94129.0 , "streetAddress" : "37000 Graham Street"}
```

Java 程序的输出和前面交互式 JavaScript shell 中的内容吻合。

现在看看同一个例子如何使用 PHP 实现。

2. MongoDB 的 PHP 驱动

首先从 MongoDB 的 github 代码库下载 PHP 驱动，配置驱动以便在本地 PHP 环境中使用。细节请参阅附录 A 中安装 MongoDB 的小节。

下面是 PHP 样例程序，它连接本地 MongoDB 服务器，列出 prefs 数据库 location 集合中的所有文档：

```
$connection = new Mongo( "localhost:27017" );
$collection = $connection->prefs->location;
$cursor = $collection->find();
foreach ($cursor as $id => $value) {
    echo "$id: ";
    var_dump( $value );
}
```

connect_to_mongodb.php

这个程序很简单，但是能完成任务！接下来，看看 Ruby 如何处理同样的任务。

3. MongoDB 的 Ruby 驱动

所有主流语言的驱动 MongoDB 都有，Ruby 也不例外。可以从 MongoDB 的 github 代码库中获取驱动，不过用 RubyGems 来管理安装更简单。要准备使用 Ruby 连接 MongoDB，最起码要获取最新的 mongo 和 bson 的 gem，安装如下：

```
gem install mongo
```

bsongem 会自动安装。此外，推荐安装 bson_ext。

代码清单 2-3 描述 Ruby 样例程序，它连接到 MongoDB 服务器并列出 prefs 数据库 location 集合中的所有文档。

代码清单 2-3 使用 Ruby 获取 MongoDB 集合中的所有文档

```
db = Mongo::Connection.new("localhost", 27017).db("prefs")
locationCollection = db.collection("location")
locationCollection.find().each { |row| puts row.inspect
```

connect_to_mongodb.rb

接下来轮到用 Python 连接 MongoDB 了。

4. MongoDB 的 Python 驱动

执行 `easy_install pymongo` 是安装 Python 驱动最简单的方式。安装好以后，可以调用代码清单 2-4 中的 Python 程序来获取 `prefs` 数据库 `location` 集合中的所有文档。

代码清单 2-4 访问 MongoDB 的 Python 程序

```
from pymongo import Connection
connection = Connection('localhost', 27017)
db = connection.prefs
collection = db.location
for doc in collection.find():
    doc
```

connect_to_mongodb.py

到这儿，同一个例子至少用了五种方式来创建和运行。这个例子简单有用，它演示了创建连接以及获取数据库、集合和集合中的文档等直接相关的概念。

2.2.2 初识Thrift

Thrift 是跨语言服务开发框架。它由一套软件和一个代码生成引擎组成，能无缝连接多种语言。Apache Cassandra 使用 Thrift 接口提供与列数据存储交互的抽象层。有关 Apache Thrift 的更多内容请访问 http://incubator.apache.org/thrift/。

Cassandra 的 Thrift 接口定义放在 Apache Cassandra 发行包中 `interface` 目录下名为 `casssandra.thrift` 的文件中。Cassandra 不同版本的 Thrift 接口定义有所不同，请确保你获得了正确版本的接口文件。此外，还要获得一份兼容定义文件的 Thrift。

Thrift 可以创建许多不同的语言绑定。在 Cassandra 的例子中，可以为 Java、C++、C#、Python、PHP 和 Perl 生成接口。生成所有 Thrift 接口最简单的命令是：

```
thrift --gen interface/cassandra.thrift
```

此外，还能为 Thrift 生成器程序声明语言参数。比如，只生成 Java Thrift 接口的命令是：

```
thrift --gen java interface/cassandra.thrift
```

生成好 Thrift 模块后，就可以在程序中使用了。假设成功生成了 Python 的 Thrift 接口和模块，就可以按照代码清单 2-5 所示，连接 `CarDataStore` 键空间查询数据。

代码清单 2-5　使用 Thrift 接口查询 CarDataStore 键空间

```
from thrift import Thrift
from thrift.transport import TTransport
from thrift.transport import TSocket
from thrift.protocol.TBinaryProtocol import TBinaryProtocolAccelerated
from cassandra import Cassandra
from cassandra.ttypes import *
import time
import pprint

def main():

  socket = TSocket.TSocket("localhost", 9160)
  protocol = TBinaryProtocol.TBinaryProtocolAccelerated(transport)
  transport = TTransport.TBufferedTransport(socket)
  client = Cassandra.Client(protocol)
  pp = pprint.PrettyPrinter(indent=2)
  keyspace = "CarDataStore"
  column_path = ColumnPath(column_family="Cars", column="make")
  key = "1"
  try:
      transport.open()
      #Query for data
      column_parent = ColumnParent(column_family="Cars")
      slice_range = SliceRange(start="", finish="")
      predicate = SlicePredicate(slice_range=slice_range)
      result = client.get_slice(keyspace,
                                key,
                                column_parent,
                                predicate,
                                ConsistencyLevel.ONE)
      pp.pprint(result)
  except Thrift.TException, tx:
      print 'Thrift: %s' % tx.message
  finally:
      transport.close()

if __name__ == '__main__':
  main()
```

query_cardatastore_using_thrift.py

尽管 Thrift 非常有用，不过有时我们还是会选择已有的语言 API。因为这些得到测试和积极支持的 API 能提供迫切需要的可靠性和稳定性，即便它们所连接的产品还处在语言的快速发展中。有许多这类 API 底层使用 Thrift。Cassandra 有很多这样的类库，比如支持 Java 的 Hector、支持 Python 的 Pycassa 和支持 PHP 的 Phpcassa。

2.3　小结

本章的目标是带你初步体验 NoSQL 数据库，通过动手实验演示核心概念。本章内容基本实

现了这一目标，并提供了超越“Hello World”的更丰富的内容。

本章通过简洁的小例子解释了 NoSQL 相关的初级概念。各种例子都从基础开始，逐渐发展到能解释一些基本概念为止。所有例子都是用领先的 NoSQL 数据库产品 MongoDB 和 Apache Cassandra 来演示的。

本章逻辑上分为两个部分：一部分介绍 NoSQL 存储的核心概念，另一部分使用主流编程语言连接 NoSQL 存储。因此，前一部分的例子都通过命令行客户端运行，而后一部分的例子可以作为单独的程序来运行。

在本章基础之上，下一章会介绍更多与 NoSQL 数据库交互和数据查询的例子，同时也会介绍其他不同的 NoSQL 新产品。

第 3 章

NoSQL 接口与交互

本章内容

- 如何访问常见的 NoSQL 数据库
- 常见 NoSQL 数据库的数据存储样例
- 常见 NoSQL 存储的集合查询
- 常见 NoSQL 数据库的驱动和语言支持

本章介绍与 NoSQL 数据存储交互的基本方式。NoSQL 存储多种多样，访问和交互方式也不尽相同。本章尝试总结 NoSQL 数据库不同访问和查询方式的最突出特点。不敢妄言覆盖全面，不过内容还算比较全面扎实。

NoSQL 技术不断发展，变化的节奏非常快。随着新的使用场景、新的编程语言和技术平台的出现，与 NoSQL 存储交互的方式也在不断变化。因而我们要做好持续学习的准备，并期待未来可能出现的标准化。

3.1 没了 SQL 还剩什么

关系型数据库普及的一个重要原因是标准化的查询和访问机制 SQL。SQL 是结构化查询语言（Structured Query Language）的简称，是关系型数据库的世界语。它的语法结构简单直观，用户在很短时间内就能熟练使用。SQL 基于关系代数，可以用来获取单表记录，或者关联多表记录。为了强调它的简单之美，这儿列举一些例子。

- 假设表 `people` 中包含某个组织中的所有人名和电子邮件地址，要获取表中所有数据，就使用 `SELECT * FROM people`。
- 表 `people` 中人名对应 `name` 列，要获取表中所有人名，使用 `SELECT name FROM people`。
- 获取所有有 Gmail 账户的人，使用 `SELECT * FROM people where email LIKE '%gmail.com'`。
- 假设表 `books_people_like` 有两列：`person_name` 和 `title`，分别存储人名（引用表 `people` 中 `name` 列里的所有数据值）和他们喜欢的书的书名，要获取相关联的人名和书名列表，使用 `SELECT people.name`、`books_people_like.book_title FROM people` 和 `books_people_like WHERE people.name = books_people_like.person_name`。

金无足赤，SQL 虽好，但也有缺点，比如处理大量稀疏数据就时常露怯。不过 NoSQL 存储里没有 SQL，更准确地说是不存在关系型数据。因此查询和访问数据的方式也有所不同。在接下来的几个小节中我们将进行揭秘，你会了解到同样是查询和访问数据，NoSQL 和 SQL 究竟有何不同，又有何相似。

首先我们来了解数据存储与访问的基本知识。

3.1.1 存储和访问数据

前一章中我们通过基本的例子初步体验了 NoSQL，主要使用的是文档存储 MongoDB 和最终一致存储 Apache Cassandra 来存储和访问数据。这一节，我们将在此基础上进一步深入了解 NoSQL 数据存储与访问。为了区分 NoSQL 不同的数据存储与访问方式，首先将它们分为下列类型。

- 文档存储：MongoDB 和 CouchDB
- 键/值存储（内存里的、可持久化的，甚至是有序的）：Redis 和 BerkeleyDB
- 列族存储：HBase 和 Hypertable
- 最终一致的键/值存储：Apache Cassandra 和 Voldermot

这个分类不是很完整，比如它忽略了对象数据库、图形数据库和 XML 数据存储，而这些都在本书范围之外。这个分类的各类之间也不是互斥的。一些 NoSQL 数据存储能提供所有分类的功能。这个分类主要还是按最恰当的描述将非关系型数据库归并成组。

> 我们对 NoSQL 存储、访问和查询的讨论仅限于所列分类，而且只考虑市面上最流行的产品。因为对于建立共通的基础 NoSQL 概念来说，几个 NoSQL 数据库足矣。对这些数据库的了解也有助于为继续阅读下面章节的高阶主题及更详尽的内容做好准备。

前面章节已经介绍过文档存储 MongoDB，下面介绍文档数据库存储和访问的细节。

实践出真知，所以还是举个例子。这个例子非常简单，但是很有趣，内容是分析 Web 服务器的日志数据。日志格式参照综合日志格式（Combined Log Format），这种格式用来记录 Web 服务器访问和请求活动的日志。更多有关 Apache Web 服务器综合日志格式的内容可以访问 http://httpd.apache.org/docs/2.2/logs.html#combined。

3.1.2 MongoDB数据存储与访问

Apache Web 服务器综合日志格式捕捉请求和响应的下列属性。

- 客户端 IP 地址：如果客户端通过代理请求资源，这个值可能是代理 IP 地址。
- 客户标识：通常不可靠，而且往往不会记录。
- 认证用户名：如果访问 Web 资源无需认证，则没有这个值。
- 请求接收时间：包括日期、时间和时区。
- 请求内容：进一步细分为四个部分：方法、资源、请求参数和协议。

- ❑ 状态码：HTTP 状态码。
- ❑ 返回对象的大小：以字节为单位。
- ❑ 提交方（Referrer）：通常是连接到 Web 页面或资源的 URI 或 URL。
- ❑ 用户代理：客户端程序，通常是访问 Web 页面或资源的程序或设备。

日志是文本文件，每个请求一行。要获取文本文件中的数据，需要解析文件并提取其中的值。用 Python 写程序解析日志文件很简单，如代码清单 3-1 所示。

代码清单 3-1　日志解析程序

```
import re
import fileinput
_lineRegex = re.compile(r'(\d+\.\d+\.\d+\.\d+) ([^ ]*) ([^ ]*)
\[([^\]]*)\] "([^"]*)" (\d+) ([^ ]*) "([^"]*)" "([^"]*)"')

class ApacheLogRecord(object):

    def __init__(self, *rgroups ):
        self.ip, self.ident, \
        self.http_user, self.time, \
        self.request_line, self.http_response_code, \
        self.http_response_size, self.referrer, self.user_agent = rgroups
        self.http_method, self.url, self.http_vers = self.request_line.split()

    def __str__(self):
        return ' '.join([self.ip, self.ident, self.time, self.request_line,
        self.http_response_code, self.http_response_size, self.referrer,
        self.user_agent])

class ApacheLogFile(object):

    def __init__(self, *filename):
        self.f = fileinput.input(filename)

    def close(self):
        self.f.close()

    def __iter__(self):
        match = _lineRegex.match
        for line in self.f:
            m = match(line)
            if m:
                try:
                    log_line = ApacheLogRecord(*m.groups())
                    yield log_line
                except GeneratorExit:
                    pass
                except Exception as e:
                    print "NON_COMPLIANT_FORMAT: ", line, "Exception: ", e
```

apache_log_parser.py

得到数据以后，把数据存储到 MongoDB 里。因为日志解析器是用 Python 写的，所以用 PyMongo（Python 的 MongoDB 驱动）把数据写入 MongoDB 更简单些。在使用 PyMongo 之前，我们先来聊聊 MongoDB 的数据存储。

> 作为文档存储的一种，MongoDB 可以存储任意数据集合，只要数据可以用 JSON 式的对象层次结构表示就行。（如果不熟悉 JSON，可以在 www.json.org/读到它的规范。这是一种在 Web 应用中流行的快速、轻量级数据互换格式。）访问日志中的一行日志用 JSON 格式可以表示如下：

```
{
    "ApacheLogRecord": {
        "ip": "127.0.0.1",
        "ident" : "-",
        "http_user" : "frank",
        "time" : "10/Oct/2000:13:55:36 -0700",
        "request_line" : {
            "http_method" : "GET",
            "url" : "/apache_pb.gif",
            "http_vers" : "HTTP/1.0",
        },
        "http_response_code" : "200",
        "http_response_size" : "2326",
        "referrer" : "http://www.example.com/start.html",
        "user_agent" : "Mozilla/4.08 [en] (Win98; I ;Nav)",
    },
}
```

对应的日志行：

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700]
"GET /apache_pb.gif HTTP/1.0" 200
2326 "http://www.example.com/start.html" "Mozilla/4.08 [en]
(Win98; I ;Nav)"
```

MongoDB 支持所有 JSON 数据类型，包括 string、integer、boolean、double、null、array 和 object。此外还支持其他一些数据类型，包括 date、object id、binary data、regular expression 和 code。之所以支持这些数据类型，是因为 Mongo 不只支持 JSON，还支持 BSON（binary encoded serialization of JSON-like structure，二进制编码序列化 JSON 结构）。有关 BSON 的规范可以访问 http://bsonspec.org/。

要在集合 `logdata` 中插入对应一行日志文件的 JSON 文档，可以用 Mongo shell 输入：

```
doc = {
    "ApacheLogRecord": {
        "ip": "127.0.0.1",
        "ident" : "-",
        "http_user" : "frank",
```

```
        "time" : "10/Oct/2000:13:55:36 -0700",
        "request_line" : {
            "http_method" : "GET",
               "url" : "/apache_pb.gif",
               "http_vers" : "HTTP/1.0",
           },
           "http_response_code" : "200",
           "http_response_size" : "2326",
           "referrer" : "http://www.example.com/start.html",
           "user_agent" : "Mozilla/4.08 [en] (Win98; I ;Nav)",
       },
   };
   db.logdata.insert(doc);
```

Mongo 还提供 save 方法，如果记录在集合中已存在则更新，不存在则插入。

用 Python 可以直接把字典（其他语言中也称为映射、哈希映射或者关联数组）表示的数据存进 MongoDB，这是因为 PyMongo（驱动）能把字典翻译成 BSON 数据格式。为了完成这个例子，创建工具函数把对象的所有属性及对应值转成字典，像下面这样：

```
def props(obj):
    pr = {}
    for name in dir(obj):
        value = getattr(obj, name)
        if not name.startswith('__') and not inspect.ismethod(value):
            pr[name] = value
    return pr
```

apache_log_parser_mongodb.py

这个函数把 request_line 看作单个元素。你可以将其保存为三个单独字段：HTTP 方法、URL 和协议版本，如代码清单 3-1 所示。还可以创建嵌套的对象层次结构，这个会在本章稍后部分讨论查询时触及。

有了这个函数，将数据存入 MongoDB 只需要几行代码：

```
connection = Connection()
db = connection.mydb
collection = db.logdata
alf = ApacheLogFile(<path to access_log>)
for log_line in alf:
    collection.insert(props(log_line))
alf.close()
```

apache_log_parser_mongodb.py

简单吧？现在你已经完成了日志数据的存储，可以对它进行过滤和分析了。

3.1.3 MongoDB数据查询

如果不能访问 Web 服务器的日志，本书提供的源码包里有一份样本数据集，是用我的 Web 服务器访问日志生成的，名为 sample_access_log。

存储完数据，就可以查询并过滤了。前面章节介绍了 MongoDB 的基本查询机制。现在回顾一下，再进一步探索与查询相关的一些补充概念。

日志数据存储在名为 logdata 的集合中。要列出 logdata 集合中所有的记录，启动 Mongo shell（JavaScript shell，可以通过 bin/mongo 命令调用），然后这样查询：

```
> var cursor = db.logdata.find()
> while (cursor.hasNext()) printjson(cursor.next());
```

数据被漂亮地展示出来，像下面这样：

```
{
    "_id" : ObjectId("4cb164b75a91870732000000"),
    "http_vers" : "HTTP/1.1",
    "ident" : "-",
    "http_response_code" : "200",
    "referrer" : "-",
    "url" : "/hi/tag/2009/",
    "ip" : "123.125.66.32",
    "time" : "09/Oct/2010:07:30:01 -0600",
    "http_response_size" : "13308",
    "http_method" : "GET",
    "user_agent" : "Baiduspider+(+http://www.baidu.com/search/spider.htm)",
    "http_user" : "-",
    "request_line" : "GET /hi/tag/2009/ HTTP/1.1"
}
{
    "_id" : ObjectId("4cb164b75a91870732000001"),
    "http_vers" : "HTTP/1.0",
    "ident" : "-",
    "http_response_code" : "200",
    "referrer" : "-",
    "url" : "/favicon.ico",
    "ip" : "89.132.89.62",
    "time" : "09/Oct/2010:07:30:07 -0600",
    "http_response_size" : "1136",
    "http_method" : "GET",
    "user_agent" : "Safari/6531.9 CFNetwork/454.4 Darwin/10.0.0 (i386)
(MacBook5%2C1)",
    "http_user" : "-",
    "request_line" : "GET /favicon.ico HTTP/1.0"
}
...
```

现在剖开查询和结果集，看看里面的细节。

首先，前面的命令声明了一个游标，接着取出 logdata 集合中的所有数据并赋值给它。游标或迭代器在关系型数据库和 MongoDB 中都很常见。

在图 3-1 中可以看到游标是如何工作的。方法 db.logdata.find()返回了集合 logdata

中的所有数据，然后用游标来遍历整个集合。代码遍历游标指向的元素并把它们打印出来。函数 printjson 用漂亮的 JSON 风格格式打印出元素，提高了可读性。

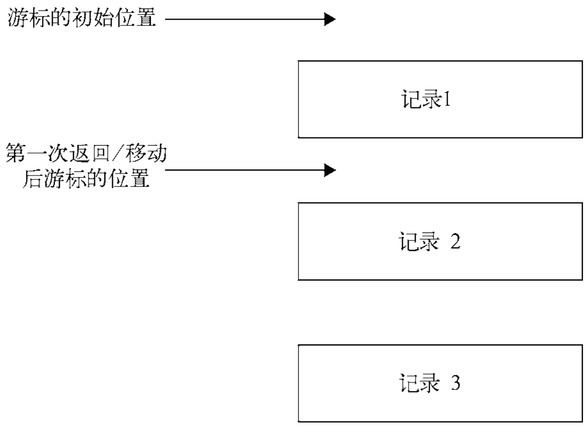


图　3-1

获取整个集合当然很好，不过通常只需要部分数据。接下来了解如何过滤集合得到子集。SQL 中获取子集常见的方式有下面两种：

- 限定输出表中部分列而非所有列；
- 限定表中行的数量，方法是根据一列或多列的值进行过滤。

在 MongoDB 里，限定输出部分列或属性并不明智，因为文档一般都是完整返回。话虽如此，真的只想获取文档的部分属性还是能做到的，只不过要限制结果集。限定文档集合为整个集合的子集与 SQL 限定结果集为指定行集类似，参见 SQL WHERE 子句。

让我们回到日志数据的例子，以展示返回集合中子集的方式。

要获取所有 http_response_code 值为 200 的日志记录，查询如下：

```
db.logdata.find({ "http_response_code": "200" });
```

这个查询接受查询文档{"http_response_code":"200"}作为 find 方法的参数，它定义了要查询的模式。

要获取所有 http_response_code 为 200，而 http_vers（协议版本）为 HTTP/1.1 的日志记录，可以这样查询：

```
db.logdata.find({ "http_response_code":"200", "http_vers":"HTTP/1.1" })
```

依然是返回一个查询文档作为 find 方法的参数，但这次传递给 find 方法的查询文档中包括两个属性。

要获取所有 user_agent 是百度搜索引擎爬虫的日志记录，可以这样查询：

```
db.logdata.find({ "user_agent": /baidu/i })
```

仔细观察语法，会注意到查询文档实际上包含的是正则表达式而非确切值。表达式/baidu/i

匹配任何 user_agent 值包含 baidu 的文档。标志 i 表示忽略大小写，所以无论 baidu、Baidu、baiDU，还是 BAIDU，所有表达式都会匹配。要获取所有 user_agent 以 Mozilla 开始的日志记录，可以这样查询：

```
db.logdata.find({ "user_agent": /^Mozilla/ })
```

查询文档支持正则表达式引发了无限的可能性，并赋予用户强大的力量。不过正如蜘蛛侠他叔伯所说：力量与责任同在。因此，尽管可以用正则表达式得到需要的子集，但要注意，复杂的正则表达式会导致成本较高的完全扫描，这对大数据来说会引发大麻烦。

数值字段可以用比较操作符，包括大于、大于等于、小于、小于等于和等于。要获取所有响应大于 1111k 的日志记录，可以这样查询：

```
db.logdata.find({ "http_response_size" : { $gt : 1111 }})
```

看过约束结果集的例子，再看看限定为一个 url 字段的例子。要查询 logdata 集合获取所有 MSN 机器人访问的 URL，命令如下：

```
db.logdata.find({ "user_agent":/msn/i }, { "url":true })
```

此外，可以将最后的查询返回的行数限制为 10：

```
db.logdata.find({ "user_agent":/msn/i }, { "url":true }).limit(10)
```

有时，只需知道匹配数量，无需返回所有文档。比如要找出 MSN 机器人请求总数，可以像下面这样查询 logdata 集合：

```
db.logdata.find({ "user_agent":/msn/i }).count()
```

有关 MongoDB 高级查询的内容还有很多，我们将留到第 6 章再讨论。接下来我们用另一个 NoSQL 存储——Redis——来存储数据。

3.1.4 Redis数据存储与访问

Redis 是持久化键/值存储。为了保持高效，它把数据库放在内存里，然后利用异步线程将其写入磁盘。它可以保存字符串、列表、哈希、集合和有序集，同时还提供了一套丰富的命令来操作集合，插入和获取数据。

安装和设置 Redis 请参阅附录 A。

下面用 Redis 命令行客户端（redis-cli）和图书分类列表的例子来解释 Redis。

第一步启动redis-cli并确认它工作正常。首先进入Redis文件目录。Redis以源码形式发布，需要解压并编译源码。编译后目录里就会出现可执行文件。在有些操作系统上，还会在公共目录中创建指向可执行程序的符号链接。我的Redis放在目录Redis-2.2.2 中，目录名对应Redis最新版本[①]。执行 redis-server 命令会启动 Redis 服务器。要使用默认配置就直接运行./redis-server。接着运行redis-cli连接服务器。Redis服务器端口默认是 6379。

① 目前 Redis 已经发布了 2.4.14 和 2.6.0-rc5 版本。——译者注

要保存键/值对`{ akey:"avalue"}`，只需键入以下内容：

```
./redis-cli set akey "avalue"
```

如果在控制台里看到响应 OK，那就没问题。如果不是，请查看安装说明并检查设置。要确认 `avalue` 已保存，可以像下面这样获取键对应的值：

```
./redis-cli get akey
```

响应是 `avalue`。

理解 Redis 样例

在这个样例中，数据库里存储的是书名列表。每本书用任意一组标签标记。例如，在列表中添加由“Michael Pollan”撰写的“The Omnivore's Dilemma”一书，并添加标签“organic”、“industrialization”、“local”、“written by a journalist”、“best seller”和“insight”。又或者添加由“Malcolm Gladwell”撰写的“Outliers”一书，并标记为“insight”、“best seller”和“written by a journalist”。之后可以获取列表中的所有书籍，或是所有标记为“written by a journalist”的书籍，又或是所有与“organic”有关的书籍，还可以获取指定作者的所有书的列表。下一节再介绍查询，现在先关注如何恰当地保存数据。

> 如果编译源代码后使用`"make install"`安装 Redis，那么 `redis-server` 和 `redis-cli` 默认会被添加到 `/usr/local/bin`。如果路径 `/usr/local/bin` 被添加到环境变量 PATH 中，你就可以从任何目录中运行 `redis-server` 和 `redis-cli` 了。

Redis 支持下列几种不同的数据结构。

- **列表，更准确地说是链表**。元素的顺序集合。访问两端速度飞快，且无关列表中元素的数量。
- **集合**。元素的无序集合，且元素之间互不重复。
- **有序集合**。元素的有序集合。
- **哈希**。键/值对的集合。
- **字符串**。字符的集合。

对这个例子，我选择用集合，因为顺序并不重要。集合命名为 `books`。每本书，即 books 集合的每个成员，有如下属性：

- Id（唯一标识）
- Title（书名）
- Author（作者）
- Tags（标签的一个集合）

每个标签有下列属性：

- Id（唯一标识）
- Name（名称）

假定 redis-server 已运行，启动 redis-cli 并输入下列命令创建 books 集合的第一个成员：

```
$ ./redis-cli incr next.books.id
(integer) 1
$ ./redis-cli sadd books:1:title "The Omnivore's Dilemma"
(integer) 1
$ ./redis-cli sadd books:1:author "Michael Pollan"
```

books_and_tags.txt

Redis 提供了一组非常有用的命令，有关它们的分类和定义可以访问：http://redis.io/commands。前面例子中第一个命令通过递增集合标识符生成了一个序列号。因为集合刚创建好，所以输出自然是“1”。接下来两个命令在 books 集合中创建了一个成员。成员标识 id 值为 1，就是刚才生成的序列号。到目前为止，成员有两个属性 title 和 author，值为字符串。命令 sadd 用来添加集合成员。列表、哈希和有序集合也有类似的命令。命令 lpush 和 rpush 分别添加一个元素到列表的头部和尾部，而命令 zadd 则添加元素到有序集合中。

接下来，为刚刚添加的 books 集合成员增加一组标签。以下是实现的命令：

```
$ ./redis-cli sadd books:1:tags 1
(integer) 1
$ ./redis-cli sadd books:1:tags 2
(integer) 1
$ ./redis-cli sadd books:1:tags 3
(integer) 1
$ ./redis-cli sadd books:1:tags 4
(integer) 1
$ ./redis-cli sadd books:1:tags 5
(integer) 1
$ ./redis-cli sadd books:1:tags 6
(integer) 1
```

books_and_tags.txt

上面给标识为 1 的成员添加了一组数字标签标识符。标签目前只定义了标识符。下面进一步分解 books:1:tags 的组成部分，解释 Redis 中键的命名体系。除了那些包含空格和特殊字符者以外，在 Redis 中任何字符串都可以作为键，但要尽量避免使用过长或过短的键。键的构成可以是结构化的，可以建立继承关系，可以嵌套对象及属性。约定俗成的键名结构是 object-type:id:field。按照这个结构，键 books:1:tags 表示集合 books 中标识为 1 的成员的标签集合。类似地，books:1:title 表示集合 books 中标识为 1 的成员的 title 字段。

给 books 集合第一个成员添加完标签后，定义标签如下：

```
$ ./redis-cli sadd tag:1:name "organic"
(integer) 1
$ ./redis-cli sadd tag:2:name "industrialization"
(integer) 1
$ ./redis-cli sadd tag:3:name "local"
(integer) 1
$ ./redis-cli sadd tag:4:name "written by a journalist"
(integer) 1
$ ./redis-cli sadd tag:5:name "best seller"
(integer) 1
$ ./redis-cli sadd tag:6:name "insight"
(integer) 1
```

books_and_tags.txt

定义完标签以后，再反过来关联包含标签的图书，建立交叉关系。第一个成员有 6 个标签，我们将其添加到每个标签中：

```
$ ./redis-cli sadd tag:1:books 1
(integer) 1
$ ./redis-cli sadd tag:2:books 1
(integer) 1
$ ./redis-cli sadd tag:3:books 1
(integer) 1
$ ./redis-cli sadd tag:4:books 1
(integer) 1
$ ./redis-cli sadd tag:5:books 1
(integer) 1
$ ./redis-cli sadd tag:6:books 1
(integer) 1
```

books_and_tags.txt

交叉关系建立好以后，可以像下面这样创建集合的第二个成员：

```
$ ./redis-cli incr next.books.id
(integer) 2
$ ./redis-cli sadd books:2:title "Outliers"
(integer) 1
$ ./redis-cli sadd books:2:author "Malcolm Gladwell"
(integer) 1
```

books_and_tags.txt

函数 `incr` 用来生成集合中第二个成员的标识。函数 `incrby` 按指定增量递增，`decr` 递减，`decrby` 按指定增量递减。如果需要生成序列号，像它们这样的函数都是非常有用的工具。可以按需选择适合的函数并定义增量，目前用 `incr` 就可以了。

接下来，为第二个成员添加标签，同时为标签建立反向关系：

```
$ ./redis-cli sadd books:2:tags 6
(integer) 1
$ ./redis-cli sadd books:2:tags 5
```

```
(integer) 1
$ ./redis-cli sadd books:2:tags 4
(integer) 1
$ ./redis-cli sadd tag:4:books 2
(integer) 1
$ ./redis-cli sadd tag:5:books 2
(integer) 1
$ ./redis-cli sadd tag:6:books 2
(integer) 1
```

books_and_tags.txt

这样就创建了两个虽然很基本但很有用的集合成员，接下来我们来学习如何查询集合。

3.1.5 Redis数据查询

我们来继续前面的 redis-cli 会话，先列出集合 books 中标识为 1 的成员的书名和作者：

```
$ ./redis-cli smembers books:1:title
1. "The Omnivore\xe2\x80\x99s Dilemma"
$ ./redis-cli smembers books:1:author
1. "Michael Pollan"
```

books_and_tags.txt

书名字符串中的特殊字符代表单引号。

列出这本书的所有标签：

```
$ ./redis-cli smembers books:1:tags
1. "4"
2. "1"
3. "2"
4. "3"
5. "5"
6. "6"
```

books_and_tags.txt

注意，列表中标签标识符的顺序和它们输入时的顺序不同，这是因为集合没有顺序。如果需要有顺序的集合，请使用有序集合。

再列出第二本书的书名、作者和标签，如下所示：

```
$ ./redis-cli smembers books:2:title
1. "Outliers"
$ ./redis-cli smembers books:2:author
1. "Malcolm Gladwell"
$ ./redis-cli smembers books:2:tags
1. "4"
2. "5"
3. "6"
```

books_and_tags.txt

现在再从标签的角度来看，列出所有拥有标签 1 的图书：

```
$ ./redis-cli smembers tag:1:books
"1"
```

标签 1 的名字是 organic，对此可以这样查询：

```
$ ./redis-cli smembers tag:1:name
"organic"
```

有些标签，例如名为 insight 的标签 6，集合中两本书都添加过。为了确认，可以查询集合中拥有标签 6 的图书，像下面这样：

```
$ ./redis-cli smembers tag:6:books
1. "1"
2. "2"
```

接下来列出所有同时拥有标签 1 和标签 6 的图书：

```
$ ./redis-cli sinter tag:1:books tag:6:books
"1"
```

命令 sinter 支持查询两个或多个集合的交集。如果“交集”一词让你觉得一头雾水，可以查看图 3-2 的文氏图。

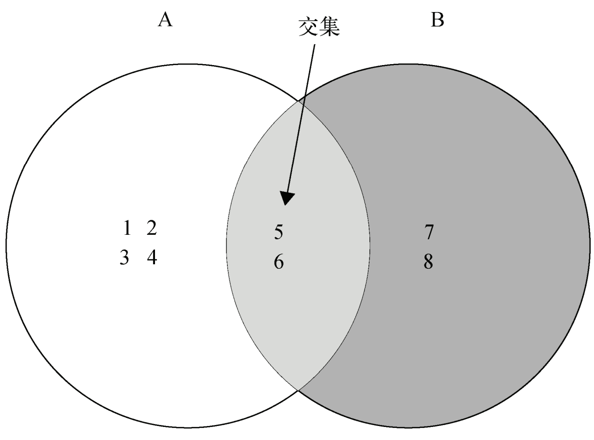


图 3-2

书 1 和书 2 都有标签 5 和标签 6，所以拥有标签 5 的图书集合和拥有标签 6 的图书集合的 sinter 应该同时列出两本书。可以运行 sinter 命令来确认，命令输入如下：

```
$ ./redis-cli sinter tag:5:books tag:6:books
1. "1"
2. "2"
```

和交集一样，还可以查询并集和差集。图 3-3 和图 3-4 演示了并集和差集的含义。

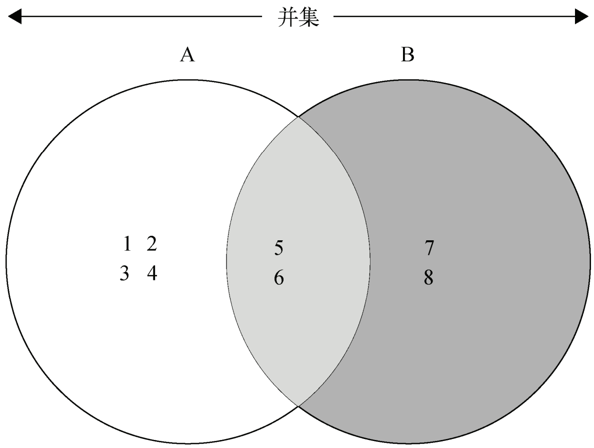


图 3-3　1，2，3，4，5，6，7 和 8（集合 A 和 B 的所有成员）同属于集合 A 和集合 B 的并集

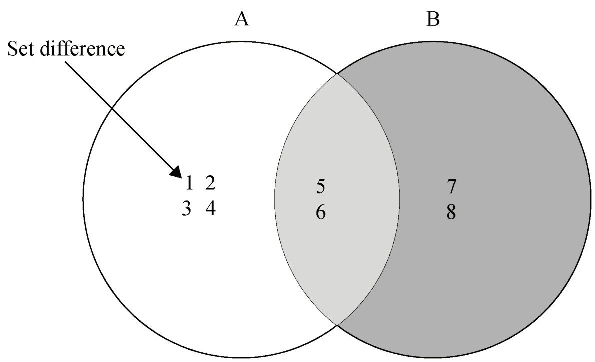


图 3-4　差集 A-B 包含了所有属于 A 但不属于 B 的成员，因此 A-B 包含 1，2，3 和 4

要创建一个包含标签 1 和标签 6 的图书并集，可以使用下列命令：

```
$ ./redis-cli sunion tag:1:books tag:6:books
1. "1"
2. "2"
```

书 1 和 2 都拥有标签 5 和标签 6，所以包含标签 5 的书和包含标签 6 的书的差集应该是空集。让我们看看是不是这样：

```
$ ./redis-cli sdiff tag:5:books tag:6:books
(empty list or set)
```

这些命令对快速查询来说是不是很有用？不仅如此，如前所述，Redis 对字符串值、列表、哈希和有序集合也支持丰富的命令。不过我们暂时先跳过这些细节，继续下一个 NoSQL 存储。所有 Redis 命令的细节都会在本书后面介绍。

3.1.6　HBase数据存储与访问

HBase 是当之无愧的 NoSQL 旗手。它是 Google Bigtable 的开源实现，关于 Google Bigtable 的更多内容请访问 http://labs.google.com/papers/bigtable.html。虽然键/值存储和其他非关系型存储（比如对象数据库）已经存在一段时间了，但真正触发 Google 式大规模 NoSQL 成功的，正是 HBase 及其相关的 Hadoop 工具。

HBase 不是唯一的类 Google Bigtable 产品，Hypertable 也算一个。HBase 也不是适用于所有情况的表格数据存储。像 Apache Cassandra 这样的最终一致性数据存储和其他许多产品所提供的功能，都超出了 HBase。不过在讨论 HBase 的适用范围之前，我们先熟悉一下 HBase 的数据存储及查询。对 HBase 和其他表格数据库适用范围的讨论会在稍后展开。

与前两个 NoSQL 存储类似，我们也用一个例子来解释 HBase 最基本的内容，更详细的架构讨论则留待第 4 章进行。这里的重点是数据存储与访问。我们先虚构一个博文 feed 的例子，博文 feed 里包含下列信息：

- 博文标题
- 博文作者
- 博文内容或主体
- 博文头部的多媒体（例如，一幅图片）
- 博文主体的多媒体（例如，一幅图片、一段视频或者一个音频文件）

为了存储这些数据，要建一个名为 `blogposts` 的集合，并把数据分成两类保存，分别为 `post` 和 `multimedia` 类。JSON 格式的条目像这样：

```
{
    "post" : {
        "title": "an interesting blog post",
        "author": "a blogger",
        "body": "interesting content",
    },
    "multimedia": {
        "header": header.png,
        "body": body.mpeg,
    },
}
```

blogposts.txt

或者也可以像这样：

```
{
    "post" : {
        "title": "yet an interesting blog post",
        "author": "another blogger",
        "body": "interesting content",
    },
    "multimedia": {
        "body-image": body_image.png,
        "body-video": body_video.mpeg,
    },
}
```

blogposts.txt

如果仔细查看两个样本数据会发现，它们都有 post 和 multimedia 类别，但不一定是同一组字段。换句话说，它们的列不同。用 HBase 的话说，它们具备相同的列族 post 和 multimedia，但是没有相同的列。实际上 multimedia 列族一共有四列，即 header、body、body-image 和 body-video，有些数据里这些列没有值（null）。在传统关系型数据库中会创建所有这四列，然后根据实际情况设置值为空。HBase 和列数据库按列存储数据，空值不需要存储，因而非常有利于保存稀疏数据集。

要创建数据集并保存这两条数据，首先启动 HBase 实例并用 HBase shell 连上它。运行在分布式环境中的 HBase 专门抽象出一层文件系统来负责保存数据到多台机器上。不过，在这个简单的例子里，HBase 运行在一个独立的单实例环境中。如果已经下载并解压了最新版的 HBase，可以通过运行 bin/start-hbase.sh 启动默认的单实例服务器。

服务器启动运行以后，启动一个 shell 连接 HBase，如下所示：

```
bin/hbase shell
```

然后创建 HBase 集合 blogposts，其中包含两个列族：post 和 multimedia，如下所示：

```
$ bin/hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.90.1, r1070708, Mon Feb 14 16:56:17 PST 2011

hbase(main):001:0> create 'blogposts', 'post', 'multimedia'
0 row(s) in 1.7880 seconds
```

添加两条数据，如下所示：

```
hbase(main):001:0> put 'blogposts', 'post1', 'post:title',
hbase(main):002:0* 'an interesting blog post'
0 row(s) in 0.5570 seconds


hbase(main):003:0> put 'blogposts', 'post1', 'post:author', 'a blogger'
0 row(s) in 0.0400 seconds

hbase(main):004:0> put 'blogposts', 'post1', 'post:body', 'interesting content'
0 row(s) in 0.0240 seconds
```

```
hbase(main):005:0> put 'blogposts', 'post1', 'multimedia:header', 'header.png'
0 row(s) in 0.0250 seconds

hbase(main):006:0> put 'blogposts', 'post1', 'multimedia:body', 'body.png'
0 row(s) in 0.0310 seconds

hbase(main):012:0> put 'blogposts', 'post2', 'post:title',
hbase(main):013:0* 'yet an interesting blog post'
0 row(s) in 0.0320 seconds

hbase(main):014:0> put 'blogposts', 'post2', 'post:title',
hbase(main):015:0* 'yet another blog post'
0 row(s) in 0.0350 seconds

hbase(main):016:0> put 'blogposts', 'post2', 'post:author', 'another blogger'
0 row(s) in 0.0250 seconds

hbase(main):017:0> put 'blogposts', 'post2', 'post:author', 'another blogger'
0 row(s) in 0.0290 seconds

hbase(main):018:0> put 'blogposts', 'post2', 'post:author', 'another blogger'
0 row(s) in 0.0400 seconds

hbase(main):019:0> put 'blogposts', 'post2', 'multimedia:body-image',
hbase(main):020:0* 'body_image.png'
0 row(s) in 0.0440 seconds

hbase(main):021:0> put 'blogposts', 'post2', 'post:body', 'interesting content'
0 row(s) in 0.0300 seconds

hbase(main):022:0> put 'blogposts', 'post2', 'multimedia:body-video',
hbase(main):023:0* 'body_video.mpeg'
0 row(s) in 0.0380 seconds
```

这两条数据分别标识为 `post1` 和 `post2`。细心的读者会注意到，在输入 `post2` 的标题时我犯了错，所以重新输入了一遍。我还重复输入了三遍同样的 `post2` 作者信息。在关系型数据库里，这些都是数据更新，不过在 HBase 里，数据不可变，重新输入数据会创建数据集的新版本。这样做有两个好处：避免了数据更新的原子性冲突；存储中内建了隐式的版本系统。

存储完数据，接着我们写些基础的查询来获取数据。

3.1.7　HBase数据查询

查询 HBase 存储最简单的办法是通过 shell。启动 `bin/hbase shell` 并连上本地 HBase，准备开始查询。

要得到所有与 `post1` 相关的数据，可以这样查询：

```
hbase(main):024:0> get 'blogposts', 'post1'
COLUMN                                              CELL
multimedia:body                                    timestamp=1302059666802,
value=body.png
multimedia:header                                  timestamp=1302059638880,
value=header.png
post:author                                        timestamp=1302059570361,
value=a blogger
post:body                                          timestamp=1302059604738,
value=interesting content
post:title                                         timestamp=1302059434638,
value=an interesting blog post
5 row(s) in 0.1080 seconds
```

blogposts.txt

结果列出 post1 的所有属性和值。要获得所有关于 post2 的数据，查询如下：

```
hbase(main):025:0> get 'blogposts', 'post2'
COLUMN                                              CELL
multimedia:body-image                              timestamp=1302059995926,
value=body_image.png
multimedia:body-video                              timestamp=1302060050405,
value=body_video.mpeg
post:author                                        timestamp=1302059954733,
value=another blogger
post:body                                          timestamp=1302060008837,
value=interesting content
post:title                                         timestamp=1302059851203,
value=yet another blog post
5 row(s) in 0.0760 seconds
```

blogposts.txt

查询只包含 post1 的标题列的列表：

```
hbase(main):026:0> get 'blogposts', 'post1', { COLUMN=>'post:title' }
COLUMN                                              CELL
post:title                                         timestamp=1302059434638,
value=an interesting blog post
1 row(s) in 0.0480 seconds
```

blogposts.txt

还记得我重新输入过 post2 的标题吗？可以同时查询这两个版本，如下所示：

```
hbase(main):027:0> get 'blogposts', 'post2', { COLUMN=>'post:title', VERSIONS=>2 }
COLUMN                                              CELL
post:title                                         timestamp=1302059851203,
value=yet another blog post
post:title                                         timestamp=1302059819904,
value=yet an interesting blog post
2 row(s) in 0.0440 seconds
```

blogposts.txt

HBase 默认只返回最新版本，不过如果你愿意，可以要求返回多个版本，或者返回某个特定的旧版本。

看过这些简单的查询后，我们进入最后一个数据存储的例子，Apache Cassandra。

3.1.8　Apache Cassandra数据存储与访问

这一节将重用上一节的 `blogposts` 样例来展示 Apache Cassandra 的一些基本特性。前面章节中，我们初步了解过 Apache Cassandra，下面继续介绍它的更多特性。

首先，在 Apache Cassandra 的目录里以前台方式启动服务器，运行命令如下：

```
bin/cassandra -f
```

服务器启动以后，运行 `cassandra-cli` 或命令行客户端，如下所示：

```
bin/cassandra-cli -host localhost -port 9160
```

查询可用的键空间，如下所示：

```
show keyspaces;
```

你会看到系统和其他已经创建好的键空间。在前一章中，我们创建了一个名为 `CarDataStore` 的键空间。现在用下面的脚本创建新的键空间 `BlogPosts`。

```
/*schema-blogposts.txt*/

create keyspace BlogPosts
    with replication_factor = 1
    and placement_strategy = 'org.apache.cassandra.locator.SimpleStrategy';

use BlogPosts;

create column family post
    with comparator = UTF8Type
    and read_repair_chance = 0.1
    and keys_cached = 100
    and gc_grace = 0
    and min_compaction_threshold = 5
    and max_compaction_threshold = 31;

create column family multimedia
    with comparator = UTF8Type
    and read_repair_chance = 0.1
    and keys_cached = 100
    and gc_grace = 0
    and min_compaction_threshold = 5
    and max_compaction_threshold = 31;
```

schema-blogposts.txt

接下来添加博文样例的数据：

```
Cassandra> use BlogPosts;
Authenticated to keyspace: BlogPosts
cassandra> set post['post1']['title'] = 'an interesting blog post';
Value inserted.
cassandra> set post['post1']['author'] = 'a blogger';
Value inserted.
cassandra> set post['post1']['body'] = 'interesting content';
Value inserted.
cassandra> set multimedia['post1']['header'] = 'header.png';
Value inserted.
cassandra> set multimedia['post1']['body'] = 'body.mpeg';
Value inserted.
cassandra> set post['post2']['title'] = 'yet an interesting blog post';
Value inserted.
cassandra> set post['post2']['author'] = 'another blogger';
Value inserted.
cassandra> set post['post2']['body'] = 'interesting content';
Value inserted.
cassandra> set multimedia['post2']['body-image'] = 'body_image.png';
Value inserted.
cassandra> set multimedia['post2']['body-video'] = 'body_video.mpeg';
Value inserted.
```

cassandra_blogposts.txt

到这里，样例就准备好了。接下来查询 BlogPosts 键空间的数据。

3.1.9 Apache Cassandra数据查询

假设你已经登录进 cassandra-cli，当前使用的是 BlogPosts 键空间，那么可以查询 Post1 的数据，如下所示：

```
get post['post1'];
=> (column=author, value=6120626c6f67676572, timestamp=1302061955309000)
=> (column=body, value=696e746572657374696e6720636f6e74656e74,
    timestamp=1302062452072000)
=> (column=title, value=616e20696e746572657374696e6720626c6f6720706f7374,
    timestamp=1302061834881000)
Returned 3 results.
```

cassandra_blogposts.txt

你也可以查询 multimedia 列族中 post2 的列 body-video。查询和输出类似下面这样：

```
get multimedia['post2']['body-video'];
=> (column=body-video, value=626f64795f766964656f2e6d706567,
    timestamp=1302062623668000)
```

cassandra_blogposts.txt

3.2　NoSQL 数据存储的语言绑定

尽管用命令行客户端访问和查询 NoSQL 数据存储非常方便快捷，但在实际应用中可能需要用编程语言接口来访问 NoSQL。NoSQL 数据存储的类型及风格不同，编程接口和驱动的类型也有所不同。不过一般情况下，对使用诸如 Python、Ruby、Java 和 PHP 这样的高级编程语言访问 NoSQL 存储都有足够的支持。本节介绍出色的代码生成器 Thrift 和一些优秀的语言特定的驱动及类库。本节的目的不在详尽覆盖所有类库，而在于帮助你掌握如何使用最喜欢的编程语言和 NoSQL 交互。

3.2.1　Thrift

Apache Thrift 是开源的跨语言的服务开发框架。它是代码生成引擎，用来创建能够与多种编程语言交互的服务。Facebook 开发了 Thrift，而后开源。

Thrift 用 C 写成。要构建、安装、使用和运行 Thrift，请按照下列步骤操作。

(1) 下载 Thrift，地址：http://incubator.apache.org/thrift/download/。

(2) 解压源代码。

(3) 按照老办法构建和安装 Thrift，即执行 configure、make 和 make install。

(4) 编写 Thrift 服务定义。这是 Thrift 中最重要的部分，它是用来生成代码的底层定义。

(5) 使用 Thrift 编译器生成特定语言的代码。

一切准备就绪。接下来，运行 Thrift 服务器并用 Thrift 客户端连上服务器。

对类似 Apache Cassandra 这样支持 Thrift 的 NoSQL 存储，或许不需要生成 Thrift 客户端，而是直接选择一个特定语言的客户端，然后客户端再利用 Thrift。下面几个小节介绍一些特定数据存储的多种语言绑定。

3.2.2　Java

作为一门“无所不在”的编程语言，Java 或许已丧失昔日的辉煌，但依旧流行或者说普及。本节介绍 MongoDB 和 HBase 的 Java 驱动及类库。

MongoDB官方支持Java驱动。下载驱动和了解更多相关内容，请访问：www.mongodb.org/display/DOCS/Java+Language+Center。这个驱动只包含一个JAR文件，编写本书时，最近的版本号是 2.5.2[①]。下载完JAR文件以后，只要把它添加到应用的classpath里就可以使用了。

前面曾在 MongoDB 实例中创建过一个 `logdata` 集合。要用 Java 驱动连上数据库并列出集合中的所有元素，可以参照代码清单 3-2 的代码实现。

① 翻译本书时，MongoDB 的 Java 驱动的最新版本是 2.8.0。——译者注

代码清单 3-2 列出 MongoDB 集合 logdata 中所有元素的 Java 程序

```
import com.mongodb.DB;
import com.mongodb.DBCollection;
import com.mongodb.DBCursor;
import com.mongodb.Mongo;

public class JavaMongoDBClient {

    Mongo m;
    DB db;
    DBCollection coll;

    public void init() throws Exception {
        m = new Mongo( "localhost" , 27017 );
        db = m.getDB( "mydb" );
        coll = db.getCollection("logdata");
    }

    public void getLogData() {
        DBCursor cur = coll.find();

        while(cur.hasNext()) {
            System.out.println(cur.next());
        }
    }

    public static void main(String[] args) {
        try{
            JavaMongoDBClient javaMongoDBClient = new JavaMongoDBClient();
            javaMongoDBClient.init();
            javaMongoDBClient.getLogData();

        } catch(Exception e) {
            e.printStackTrace();
        }

    }

}
```

javaMongoDBClient.java

使用 Java 与 MongoDB 交互非常简单方便，不是吗?

接下来是 HBase。要查询 HBase 中已经创建好的 `blogposts` 集合，首先找到下列 JAR 文件并把它们添加到 classpath 中。

- commons-logging-1.1.1.jar
- hadoop-core-0.20-append-r1056497.jar
- hbase-0.90.1.jar
- log4j-1.2.16.jar

代码清单 3-3 的程序列出了 `blogposts` 中 `post1` 的标题和作者。

代码清单 3-3　连接和查询 HBase 的 Java 程序

```
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.io.RowResult;

import java.util.HashMap;
import java.util.Map;
import java.io.IOException;

public class HBaseConnector {

public static Map retrievePost(String postId) throws IOException {
HTable table = new HTable(new HBaseConfiguration(), "blogposts");
Map post = new HashMap();

RowResult result = table.getRow(postId);

for (byte[] column : result.keySet()) {
post.put(new String(column), new String(result.get(column).getValue()));
}
return post;
}

public static void main(String[] args) throws IOException {
Map blogpost = HBaseConnector.retrievePost("post1");
System.out.println(blogpost.get("post:title"));
System.out.println(blogpost.get("post:author"));
}
}
```

HBaseConnector.java

看过 Java 的两个代码样例后，我们继续进入下一种编程语言：Python。

3.2.3　Python

前面 MongoDB 的日志数据样例中曾用过 Python，这里再介绍另外一个 Python 的例子。这一次，用 Pycassa 来实现 Python 与 Apache Cassandra 的交互。

首先，从 http://github.com/pycassa/pycassa 获取 Pycassa 并安装到本地的 Python 环境中。安装完成后，像下面这样导入 Pycassa：

```
import pycassa
```

接着，连接本地 Cassandra 服务器（一定记得先启动服务器）。像这样连接服务器：

```
connection = pycassa.connect('BlogPosts')
```

连上以后，可以获取 `post` 列族：

```
column_family = pycassa.ColumnFamily(connection, 'post')
```

然后调用 `get()` 方法来获取列族中所有列的数据：

```
column_family.get()
```

可以把行键作为参数传入 get()方法，这样可以只输出行键对应的列。

3.2.4 Ruby

针对 Ruby，我选择用 Redis 客户端的例子来介绍。Redis 数据集里包含了图书和关联的标签。首先，复制 redis-rb git 库，构建 redis-rb，如下所示：


```
git clone git://github.com/ezmobius/redis-rb.git
cd redis-rb/
rake redis:install
rake dtach:install
rake redis:start &
rake install
```

或者也可以这样安装 Redis gem：

```
sudo gem install redis
```

安装好 redis-rb 以后，打开一个 irb（interactive ruby console，交互式 ruby 命令行）会话，然后连接和查询 Redis。

首先，使用 require 命令导入 rubygems 和 Redis，如下所示：

```
irb(main):001:0> require 'rubygems'
=> true
irb(main):002:0> require 'redis'
=> true
```

接着，确认 Redis 服务器正在运行，然后实例化一个 Redis 对象来连接服务器，如下所示：

```
irb(main):004:0> r = Redis.new
=> #<Redis client v2.2.0 connected to redis://127.0.0.1:6379/0 (Redis v2.2.2)>
```

接下来，列出 books 集合中标识为 1 的图书的所有标签：

```
irb(main):006:0> r.smembers('books:1:tags')
=> ["3", "4", "5", "6", "1", "2"]
```

列出同时拥有标签 5 和标签 6 的所有图书（参考本章前面的例子）：

```
irb(main):007:0> r.sinter('tag:5:books', 'tag:6:books')
=> ["1", "2"]
```

除此之外，还能运行许多更高阶的查询，步骤都非常简单。下面要介绍的最后一个例子使用 PHP 来与 NoSQL 交互。

3.2.5 PHP

Pycassa 是在 Thrift 上提供了一层 Python 封装，和它一样，phpcassa 在 Thrift 上提供了一层 PHP 封装。可以在 http://github.com/hoan/phpcassa 下载 phpcassa。

有了 phpcassa，要查询 BlogPosts 集合 post 列族的所有列只需几行代码，像下面这样：

```
<?php  // Copy all the files in this repository to your include directory.
$GLOBALS['THRIFT_ROOT'] = dirname(__FILE__) . '/include/thrift/';
require_once $GLOBALS['THRIFT_ROOT'].'/packages/cassandra/Cassandra.php';
require_once $GLOBALS['THRIFT_ROOT'].'/transport/TSocket.php';
require_once $GLOBALS['THRIFT_ROOT'].'/protocol/TBinaryProtocol.php';
require_once $GLOBALS['THRIFT_ROOT'].'/transport/TFramedTransport.php';
require_once $GLOBALS['THRIFT_ROOT'].'/transport/TBufferedTransport.php';
include_once(dirname(__FILE__) . '/include/phpcassa.php');
include_once(dirname(__FILE__) . '/include/uuid.php');

$posts = new CassandraCF('BlogPosts', 'post');
$posts ->get();
?>
```

phpcassa_example.php

学习完这个例子之后，我们来回顾一下本章所学的内容，并继续了解有关 NoSQL schema 的更多内容。

3.3 小结

本章介绍了 NoSQL 交互、访问与查询的基本概念。样例中用到了四种具有代表性的 NoSQL 存储：MonogDB、Redis、HBase 和 Apache Cassandra。通过样例演示了如何用它们存储、访问类哈希结构或表格结构的数据。

之后解释了各种查询方法，它们大都用简单的命令行客户端完成。最后几节介绍了语言绑定以及 Java、Python、Ruby 和 PHP 客户端类库。对类库的覆盖还不够详尽，不过足以入门，而且能满足大多数基本操作的需要。接下来一章将介绍 NoSQL 结构以及元数据的相关概念和知识。

Part 2 第二部分

NoSQL 基础

本部分内容

第 4 章

理解存储架构

本章内容

- 面向列数据库的存储架构
- 文档数据库内部机制
- 键/值缓存和磁盘存储的键/值
- 支持最终一致性的列族数据集的 schema

面向列数据库是最流行的一种非关系型数据库。Google 令人敬佩的工程努力使其一举成名，而 Facebook、LinkedIn 和 Twitter 等社交网络巨头的发展则使它们更加流行，说它们是 NoSQL 革命的旗手，一点都不夸张。尽管在学术界列数据库已经存在多年，形式多样，不过直到以下几篇 Google 研究论文的发表，它们才被引入开发社区。

- The Google File System。http://labs.google.com/papers/gfs.html（2003 年 10 月）。
- MapReduce: Simplified Data Processing on Large Clusters。http://labs.google.com/papers/mapreduce.html（2004 年 12 月）。
- Bigtable: A Distributed Storage System for Structured Data。http://labs.google.com/papers/bigtable.html（2006 年 11 月）。

这些论文介绍了 Google 搜索引擎的成功秘诀，阐明了诸如 Google Earch、Google Analytics 和 Google Maps 等大规模及大数据产品背后使用的技术。过去人们怀疑，能否用大量廉价硬件组成集群来存储海量数据（远远超过一台机器的承载量），同时保证数据在合理的时间范围内得到高效处理。Google 的论文回答了人们的疑问，其中包括 3 个关键主题：

- 数据存储应使用网络文件系统，系统要能扩展到多台机器上。文件本身可能非常大，得存储在多个节点上，每个节点运行在不同的机器上。
- 数据存储结构应具备多种特性。比传统的规范化关系型数据结构具有更好的灵活性，能高效存储海量稀疏数据，能够适应结构的不断调整，同时还不用更改底层表。
- 数据处理方式应支持在互相隔离的数据子集上执行计算，然后再合并计算结果，最后生成所需要的输出。如果算法放到数据所在地去执行，就能提高计算效率。当处理海量数据运算时，还能避免在网络上传输大量数据。

Google 分享的内容和智者之言，如同雨露浇灌着开源的沃土，大量开源产品如雨后春笋般破土而出，诞生出了一批令人瞩目的面向列数据库产品。其中最有名的非 Apache Hadoop 莫属，它

如同一面镜子，映射出了Google基础架构的完整轮廓。2004年至2006年间，开源搜索引擎Lucene和Nutch的创始人Doug Cutting发起了Hadoop项目，早期目的是要解决在开发Nutch过程中遇到的扩展性问题。后来，在Yahoo!的工程师、一些开源贡献者以及Hadoop早期用户的支持下，Hadoop逐渐成熟，变成了一个可以在生成环境中正式使用的平台。与此同时，NoSQL运动也不断积蓄力量，涌现出大量代替Hadoop的选择，有些还在原有模型基础上进行了改进。这些产品中，大多数都没有重新发明网络文件系统，或是数据处理的方法论，而是在列数据存储的功能列表上，添加了属于自己的功能。在下面的章节里，我们会学习这些面向列数据库的底层基础知识。

> Doug Cutting 的演示稿中记录了 Hadoop 的发展简史，要获取该文档请访问：http://research.yahoo.com/files/cutting.pdf。

4.1　使用面向列的数据库

Google Bigtable和Apache HBase（Hadoop的一部分）都属于面向列数据库，Hypertable和Cloudata也是如此。它们各不相同，但有着共同的基础。本节中，我会解释这些定义它们的基础概念。

在今天这一代开发者的大脑里，关系型数据库系统的概念早已根深蒂固。关系型数据库管理系统（RDBMS）是我们在大学里学到的、工作中用到的、经常读到和讨论的概念，其中最根本的概念（比如实体和它们之间的关系）已经成为数据库概念不可分割的组成部分。所以，我们首先从RDBMS的视角出发，来解释面向列数据库，这样大家会感觉舒适自然。然后，我们再从映射表也就是键/值对的角度，重新解读一遍。

4.1.1　使用关系型数据库中的表格和列

在RDBMS里，实体的属性存储在表格的列中。列预先定义好，元素或行记录的所有值存储在表格的所有列中。如图4-1所示。

这个例子中的表格有5列。在RDBMS中存储这个表时，要定义每列的数据类型。比如，名字列设为VARCHAR类型（变长字符类型），而ZIP列设为整数类型（在美国所有ZIP代号都是整数）。单元格可设空值（即NULL）。例如，Jolly Goodfellow这个名字里没有中间名（middle name），则中间名一列的值为NULL。

通常情况下，一个RDBMS表格会有不少列，有时有几十列，表格里可多达几千条记录。有时表格里保存了数百万行记录，但是保存这么大量的数据可能导致数据访问阻塞，除非引入一些特殊的做法，比如去规范化。

表格里存进数据以后，往往需要修改表格，新增一些属性。这些属性可能是街道地址，或者食物喜好。添加完新属性，再存储新数据，新属性带上了值，但是老记录里新属性的值可能就是null。不仅如此，属性种类越多，出现稀疏数据集（大量单元格值为 null）的可能性就越大。直

到某个时刻，整个表可能会变成像图 4-2 这样。

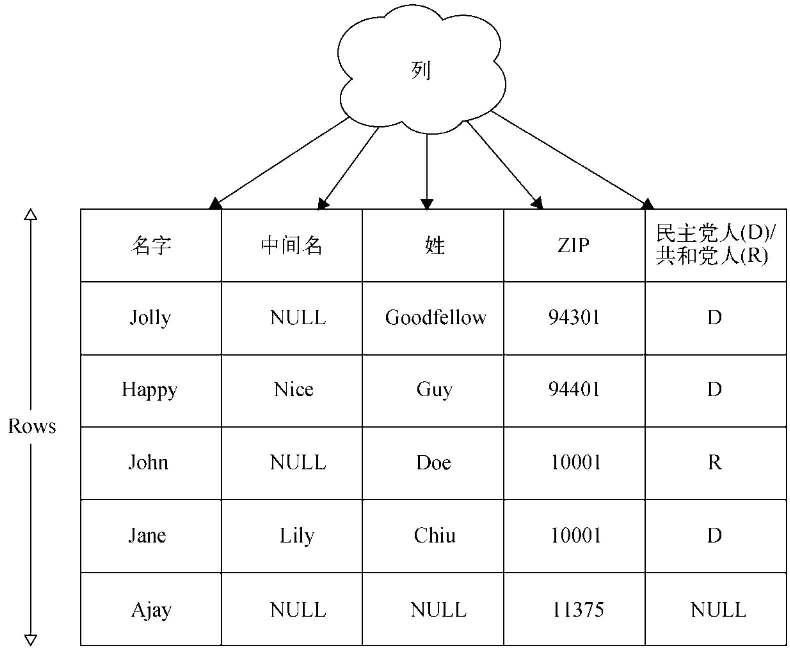


图　4-1

名字	中间名	姓	街道地址	ZIP	民主党人/共和党人	素食主义者/非素食主义者

图　4-2

现在，假设数据不断变化，而你要随之保存单元格每个版本的值。想象一个三维 Excel 表格，第三维是时间。值的每个版本依次放入按时间排列的表格。浏览一下图 4-3，让自己沉浸于三维 Excel 表格的抽象之中吧。

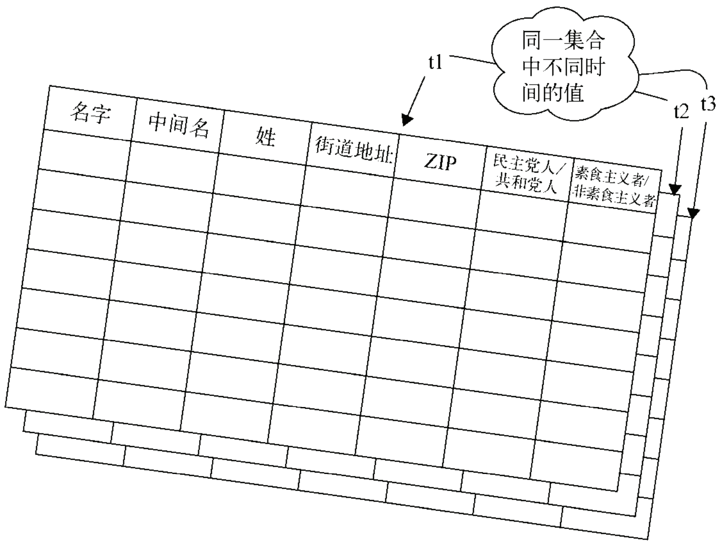


图 4-3

虽然这个例子很简单，有人可能已经意识到，按照数据的变化修改表结构，存储大量稀疏数据，使用不同版本的值，这些事情可能很复杂。更准确地说，用 RDBMS 处理可能变得很复杂！这些困难你大抵也经历过吧。

4.1.2 列数据库对比RDBMS

下面，介绍用列数据库来完成同一个例子的建模和存储。前面有了 RDBMS，两相比较，列数据库的主要特点自然会更加明显。

首先，面向列数据库最小化了前期 schema 定义的需求，后期随着数据的发展，可以很容易地添加新列。在一般的面向列数据库中，预先定义列族而不是列。列族是成组的列，各成员逻辑上相互关联，但无强制要求。列族成员物理上存储在一起，访问特征相似的列加入同一个列族对用户是有好处的。可定义的列族总数，就算理论上存在限制，也不会很大，不过控制列数，保持最小量，有助于维持 schema 的可延展性。在这个例子里，定义姓名、地址、偏好三个列族就够了。

列数据库里的列族与 RDBMS 里的列相似。两者都是在数据存储到表里之前就定义好了，并且本质上相对保持静态。RDBMS 里的列定义了要存储的数据类型，而列族没有。列族可以包括任意多个列，列可以存储任意类型数据，只要数据可以保存为字节数组就行。

面向列数据库表里，只有在列值有效时，才会存储到行中。Null 值不会被存储。下图 4-4 中，为了适应列存储模型，对例子做了适当的修改。

	姓名	地址	偏好
	first name=>"…", last name=>"…"	zip=>"…"	d/r=>"…", veg/non-veg=>"…"

图　4-4

作为存储容器，列数据库除了对稀疏的和可变的数据友好，还为每个单元格的数据保存多个版本。因此，如果不断地改变样例数据，数据就会存储到列数据库中，如图 4-5 所示。

	时间	姓名	地址	偏好
	t9	first name=>"…", last name=>"…"	zip=>"…"	d/r=>"…", veg/non-veg=>"…"
	t8			
	t7			
	t5			

图　4-5

在物理存储方面，数据并不是按表存储的，而是按列族存储的。列数据库是为可扩展性设计的，可以轻松容纳数百万列和数十亿行。因此，单表常常存储在多台机器上。行键唯一标识列数据库中的一行记录。行记录是有序的，随着增长，数据会被分割成组，组里包含了连续的值。图 4-6 对数据的实际存储方式做了非常接近的描述。

列数据库通常部署在集群上，不过为了开发和实验，也可以在单节点上运行列数据库。不同的列数据库一般都有不同的网络拓扑和部署架构，但是只要深入学习其中任何一个，就可以了解其余了。

> 本章稍后的 4.2 节中将介绍 HBase 架构，讲述更多有关典型部署的内容。

行键	时间	姓名
	t9	
	t8	
	t7	
	t5	

行键	时间	地址
	t9	
	t8	

行键	时间	偏好
	t9	
	t7	

图　4-6

4.1.3　列数据库当做键/值对的嵌套映射表

把列数据库当成具有特殊属性的表，虽然很好理解，但是容易产生混淆。列、表这样的术语能让人瞬间想到关系型数据库知识，进而按此规划 schema。要真这么做，那就算是被带进沟里去了，而且常常反复往沟里跳，直把列数据库当关系型数据库使，还乐此不疲。这正是一个需要避免的设计误区。时刻谨记，使用正确的工具解决问题比工具本身更重要。如果 RDBMS 符合需要，那就用它。但如果要用列数据库扩展数据存储，那就别背上 RDBMS 的包袱。

通常，把列数据库看成嵌套的映射表更容易理解。映射表，或哈希映射表，又称关联数组，是键及其对应值组成的二元组。键要保持唯一以避免冲突，值可以是任何字节数组。有些映射表只存储字符串类型的键/值，不过大多数列数据库没有这种限制。

现代列数据库的灵感来源于 Google BigTable，其官方定义为稀疏的、分布式的、持久化的、多维的有序映射表。

> "Bigtable: A Distributed Storage System for Structured Data," Fay Chang, et al. OSDI 2006, http://labs.google.com/papers/bigtable-osdi06.pdf. 第二节"数据模型"里这样定义 Bigtable："Bigtable 是稀疏的、分布式的、持久化的、多维的有序映射表。映射表的索引包括行键、列键和时间戳；映射表中的每个值都是原始的字节数组。"

来看一个多维嵌套映射表的例子，头两层键用类 JSON 形式表示如下：

```
{
  "row_key_1" : {
    "name" : {
    ...
    },
    "location" : {
    ...
    },
    "preferences" : {
    ...
    }
  },
  "row_key_2" : {
    "name" : {
    ...
    },
    "location" : {
    ...
    },
    "preferences" : {
    ...
    }
  },
  "row_key_3" : {
  ...
}
```

最外层的键是行键，唯一标识列数据库中的一条记录。第二层键是列族标识。前面一共定义了三个列族，包括姓名、位置和偏好。它们都是第二层键。如此下去，你可能已经猜到第三层键是列标识符。由于各行拥有（同一列族的）的列可能不同，因此任意两行数据第三层的键可能会不同。加入第三层键，映射表看上去就会像这样：

```
{
  "row_key_1" : {
    "name" : {
      "first_name" : "Jolly",
      "last_name" : "Goodfellow"
      }
```

```
        }
      },
      "location" : {
        "zip": "94301"
      },
      "preferences" : {
        "d/r" : "D"
      }
    },
    "row_key_2" : {
      "name" : {
        "first_name" : "Very",
        "middle_name" : "Happy",
        "last_name" : "Guy"
      },
      "location" : {
        "zip" : "10001"
      },
      "preferences" : {
        "v/nv": "V"
      }
    },
    ...
  }
```


最后，再加入版本，第三层可以包含时间戳版本号。为了表示这个版本，使用任意整数来表示时间戳版本号，然后编个故事，说 Jolly Goodfellow 在时刻 1 宣布了他的民主倾向，然后在时刻 5 又改变他的政治派别为共和党。这行数据的映射表看起来像这样：

```
{
  "row_key_1" : {
    "name" : {
      "first_name" : {
        1 : "Jolly"
       },
      "last_name" : {
        1 : "Goodfellow"
      }
    },
    "location" : {
      "zip": {
        1 : "94301"
      }
    },
    "preferences" : {
      "d/r" : {
        1 : "D",
        5 : "R"
      }
    }
  },
  ...
}
```

以上展现的是从映射表角度看列数据库时所呈现出的画面。如果觉得样例不够详细，还可以阅读 Jim Wilson 的文章，标题是“Understanding HBase and Bigtable”（理解 HBase 和 Bigtable），地址是 http://jimbojw.com/wiki/index.php?title=Understanding_Hbase_and_BigTable。

4.1.4　Webtable布局

Webtable 用来存储抓取到的网页副本，如果放着这个经典案例不谈，那对列数据库的讨论肯定是不完整的。Webtable 表格中存储网页内容以及页面相关属性。这些属性可以是引用页面的锚点，或者内容的 MIME 类型。这个例子第一次出现是在 Google 有关 Bigtable 的研究论文里。Webtable 使用网页的反向 URL 作行键。若 URL 为 www.example.com，则对应行键 com.example.www。面向列数据库中行键决定了行的顺序。所以当反向 URL 作行键时，与 example.com 的两个子域名（比如 www.example.com 和 news.example.com）相关的行集会存储在彼此邻近的位置上。这样查询和某个域名相关的内容就更容易一些。

通常情况下，内容、锚点和 MIME 类型是列族，其概念模型类似基于列的表格，如图 4-7 所示。

行键	时间	内容	锚点	mime
com.cnn.www	t9		cnnsi.com	
	t8		my.look.ca	
	t6	"<html>…"	my.look.ca	"text/html"
	t5	"<html>…"		
	t3	"<html>…"		

图　4-7

Bigtable 的许多开源实现在文档中都包含了 Webtable 的例子。例如 HBase 架构的维基条目，地址：http://wiki.apache.org/hadoop/Hbase/HbaseArchitecture；Hypertable 数据模型文档，地址：http://code.google.com/p/hypertable/wiki/ArchitecturalOverview#Data_Model。

概述了列数据库的概念之后，下面介绍 HBase 的部署和存储模型。在列数据库中 HBase 的分布式部署模型堪称典型，对了解 Web 规模数据库架构是一个很好的起点。

4.2 HBase 分布式存储架构

一个健壮的 HBase 架构除了 HBase 本身，还包括其他一些部分，至少得包括一个分布式的中央服务，用来进行配置和同步。架构总览如图 4-8。

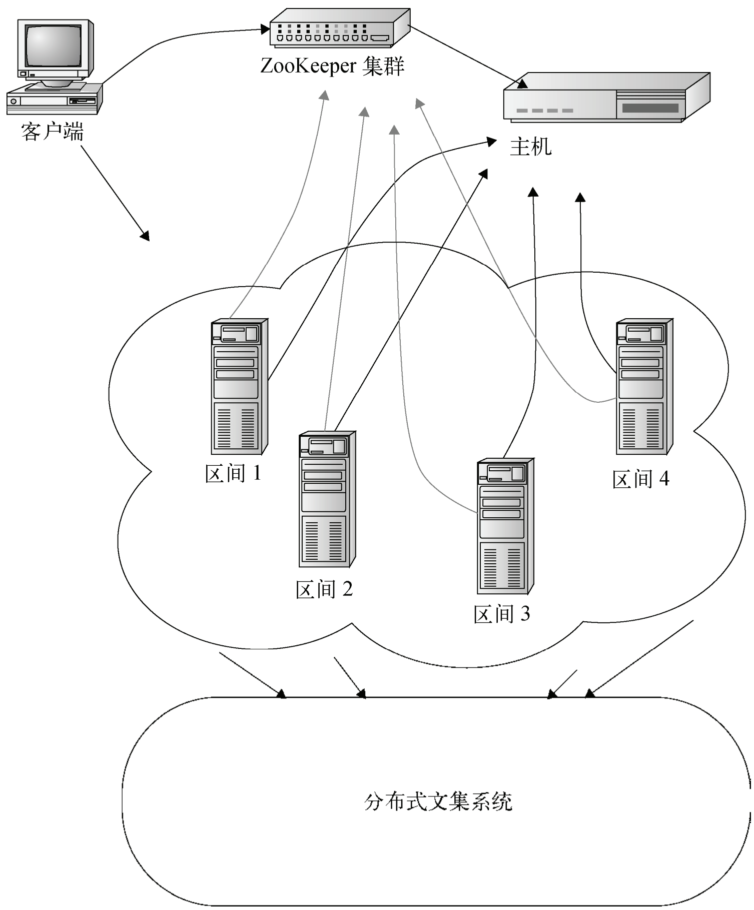


图 4-8

HBase 的部署遵从主从模式。通常有一个主机和一组从机，俗称 range server。启动时，主机给每台 range server 分配一组区间。每个区间包含一个行记录的有序集合，行记录由行键唯一标识。如果存储在区间里的行记录数量超过了配置的阈值，区间就会分割成两个新的区间，行记录在这两个新的区间之间分配。

和大多数列数据库一样，HBase 中列族的所有列存放在一起。因此每个区间为每张表的每个列族维护独立的存储。存储和底层分布式文件系统的物理文件一一对应。对这些单独的存储，

HBase 封装了一层很薄的 API，抽象掉了对底层文件系统的访问，存储用它作为中介访问底层物理文件。

每个区间都有内存存储（或者说是缓存）和 WAL（write-ahead-log，预写日志）。下面引述维基百科（http://en.wikipedia.org/wiki/Write-ahead_logging），WAL 是在数据库系统中提供原子性和持久性（ACID 的两个属性）的一族技术。WAL 常见于各种数据库系统，包括流行的关系型数据库系统，例如 PostgreSQL 和 MySQL。在 HBase 中，客户端程序能决定是否开启 WAL。关闭 WAL 能提升性能，但在出现错误时会降低可靠性和恢复能力。如果开启了 WAL，数据写入区间前会先写日志，然后写入区间的内存存储。内存存储填满以后，数据会刷入磁盘，持久化到底层分布式存储中。region server 和区间的核心概述如图 4-9 所示。

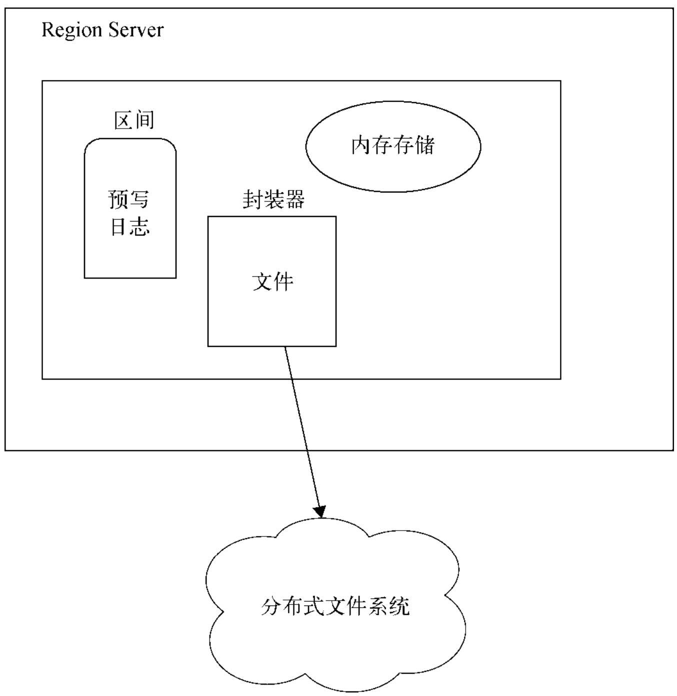


图　4-9

如果使用了类似 HDFS（Hadoop Distributed FileSystem，Hadoop 分布式文件系统）的分布式文件系统，那么主从模式还会扩展到底层存储架构。HDFS 里，一个 namenode 和一组 datanode 形成的结构非常类似于 HBase 等列数据库中主服务器和 range server 的配置。这种情况下，HBase 里每个列族的物理存储文件最终落到 HDFS 的 datanode 上。HBase 利用文件系统 API 避免了与 HDFS 强耦合，所以文件系统 API 可以看成是 HBase 和相应的 HDFS 文件相交流的中介。这个 API 还支持 HBase 无缝地集成其他类型的文件系统。比如 HBase 可以使用 CloudStore，它的前身

是 KFS（Kosmos FileSystem，Kosmos 文件系统），而非 HDFS。

> 更多有关 CloudStore（前身是 KFS）的内容请访问 http://kosmosfs.sourceforge.net/。

除了利用分布式文件系统做存储，HBase 集群还利用到了外部配置与协调工具。在 Bigtable 的论文里，Google 将这个配置程序命名为 Chubby。作为 Google 基础设施的镜像，Hadoop 也创造了一个正好对应的产品，名为 ZooKeeper。Hypertable 也有一个类似的基础设施部件，名为 Hyperspace。ZooKeeper 集群通常用作 HBase 集群的前端，负责处理新客户和管理配置。

第一次访问 HBase 时，客户端先通过 ZooKeeper 访问两个目录-ROOT-和.META。这些目录维护着所有区间的状态和位置信息。-ROOT-维护和所有.META 表相关的信息，.META 文件维护和用户空间表（即保存数据的表）相关的信息。客户端想要访问某行数据时，首先问 ZooKeeper 要-ROOT-目录。然后从-ROOT-目录定位到行记录的.META 目录，.META 目录提供了和行记录有关的所有区间的明细，最后用这个信息访问行记录。下次再访问同一行数据时，客户端不会再重复这个三步查找过程。列数据库非常依赖对三步查找过程相关信息的缓存，这样下次客户端需要行数据时，就可以直接访问 region server。只有在区间缓存过期、区间被禁用或者区间无法访问的情况下，才会重复查找循环。

每个区间通常用存储的最小行键来标识，所以查找行记录非常简单，检查行键大于等于区间标识符即可。

到这儿，列数据库存储的基本概念和物理模型都已经介绍完了，它们读写数据的机制也公开了。后面章节里还会介绍一些相关的高阶特性和细节的微小差别，下面我们转向文档存储。

4.3 文档存储内部机制

前面章节从用户角度介绍了文档存储 MongoDB，下一步我们“开剥洋葱皮”。

MongoDB 是文档存储，文档按组分成集合。在概念上，可以认为集合类似关系表。不过集合并不对 schema 进行严格的约束，这点和关系表不同。一个集合可以包含任何文档，不过为了便于进行有效的索引，同一个集合的文档还是相似更好。集合用命名空间隔离，但内部形式不是按命名空间来划分层次的。

文档存储为 BSON 格式。BSON 是 JSON 类文档的二进制编码形式，结构类似嵌套键/值对。BSON 是 JSON 的超集，额外支持一些类型，比如正则表达式、二进制数据和日期。每个文档都有一个唯一标识符，如果数据插入集合前没有明确指定唯一标识符，可以由 MongoDB 生成，自动生成的对象 ID 结构如图 4-10。

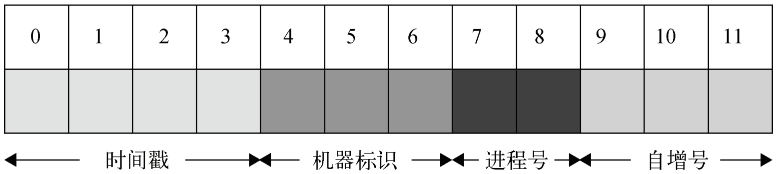


图　4-10

MongoDB 的驱动和客户端在访问 BSON 编码的数据时，会对数据进行序列化和反序列化操作。另一方面，MongoDB 服务器理解 BSON 格式，无需序列化，因而避免了额外的开销。数据在传输和读取过程中，都采用相同的二进制格式，这极大地提升了性能。

BSON 像 Protocol Buffers 吗？

Protocol Buffers，有时又称 protobuf，是 Google 为了实现高效传输而对结构化数据进行编码的方式。Google 用它来进行所有内部的远程调用（Remote Procedure Call，RPC）和交换格式。Protobuf 和 XML 类似，是结构化格式，但是比 XML 更轻、更快、更高效。Protobuf 是语言和平台中立的规范和编码机制，可以在许多语言中使用。更多有关 protobuf 的内容请访问：http://code.google.com/p/protobuf/。

BSON 和 protobuf 的相似之处在于它也是语言和平台中立的编码机制、数据交互格式和文件格式。与 protobuf 相比，BSON 结构化更弱，这样增强了灵活性，但也失去了明确定义 schema 的性能优势。尽管 BSON 和 MongoDB 有关联，但是完全可以脱离 MongoDB 使用这种格式。MongoDB 驱动主要用于和 MongoDB 服务器交互，但是 BSON 序列化特性可以单独利用。更多有关 BSON 的内容请访问：http://bsonspec.org/。

高性能是贯穿 MongoDB 设计的一个重要理念。使用内存映射文件做存储正好体现了这种设计理念。

4.3.1　用内存映射文件存储数据

内存映射文件是一段逐字节分配给文件的虚拟内存，或者是能通过文件描述符引用的一种类似于文件的资源。这样，应用程序和这些文件交互时，可以把它们看作主内存的一部分。与一般的磁盘读写相比，这可以明显提高 I/O 性能。访问和操作内存比系统调用快多了。除此以外，诸如 Linux 等操作系统，映射到文件的内存区域属于内存页缓存的一部分。这个缓存对应用完全透明，通常称为页缓存，由操作系统的内核来实现。

MongoDB 选择用内存映射文件存储，这种策略很聪明，但是也造成了一些影响。首先，内存映射文件暗示没有分离操作系统缓存和数据库缓存，也没有缓存冗余。其次，由于虚拟内存映射并非在所有操作系统上都以同样的方式工作，所以缓存受操作系统控制：至于什么能保存在缓存里，什么会被丢弃掉，缓存管理策略会随操作系统不同而变化。第三，无需任何额外配置，MongoDB 的数据库缓存就能用掉所有可用内存，这说明通过提供更大的 RAM 和分配更大的虚

拟内存可以提升 MongoDB 的性能。

内存映射也有一些限制。比如 MongoDB 在 32 位系统上数据最大值不能超过 2GB。64 位机器上则没有这个限制。

数据库大小不是唯一的限制。其他限制包括文档大小和 MongoDB 服务器可以容纳的集合数量。单个文档不能超过 8MiB，这说明不适合用 MongoDB 存储大对象。如果必须存储大于 8MiB 的文档，就要用 GridFS。此外，一个数据库实例默认支持 24 000 个命名空间。每个集合和索引各占一个命名空间，即如果每个集合默认有两个索引，则一个数据库最多容纳 8000 个集合。这么大的数量通常足够了，如果还需要更多，可以增加命名空间数量，使其超过 24 000。

增加命名空间数量也有限制和影响。每个集合命名空间都要占用几 kB。MongoDB 的索引用 B 树实现，每个 B 树页占 8kB。如果增加了额外的命名空间，无论对集合还是索引，每个额外的实例都要多占用几 kB。数据库 mydb 的命名空间保存在文件 `mydb.ns` 中。像 `mydb.ns` 这样的 `.ns` 文件最大不能超过 2GB。

由于大小上的约束限制了数据库无限增长，所以了解集合和索引的行为模式就非常重要。

4.3.2 MongoDB集合和索引使用指南

没有公式能确定数据库中集合数量的最佳值，所以最好避免在一个集合中放入过多不同种类的数据。不同来源的数据混合在一起会增加索引的复杂性。一个好的经验法则是先问问自己是否经常需要跨集合查询。如果答案是肯定的，那就把数据放在一起，否则就分到不同集合，因为那样效率更高。

有时候集合可能无限增长，直逼 2GB 数据库大小。这种情况下可以考虑用定量集合。MongoDB 定量集合像是预定义好大小的栈。定量集合达到预定义大小后，旧数据会被删除。旧数据用 LRU（Least Recently Used，最近最少使用）算法确定。从定量集合中获取文档服从 LIFO（Last-In-First-out，后进先出）原则。

> 了解更多有关 LRU 算法的内容，请访问：
> http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used。

所有 MongoDB 集合的`_id` 字段都有索引。此外，文档的任何属性上都可以定义索引。查询时，集合文档按集合中`_id` 的自然顺序返回。只有定量集合使用 LIFO 顺序（即插入顺序）。游标分批返回数据，每批限制在 8MiB 以内。记录更新原地进行。

MongoDB 提供了增强的性能，但其代价是牺牲可靠性。

4.3.3 MongoDB的可靠性和耐久性

MongoDB 并不总是注意原子性，也没定义并发操作中的事务完整性和隔离级别。因此在更新同一个集合时，两个进程可能相互冲突。只有称为 Modifier Operation 的特定一类操作才提供

了原子性。

> MongoDB 为原子更新定义了如下一些 modifer operation。
> - `$inc`：累加字段值
> - `$set`：设定字段值
> - `$unset`：删除字段
> - `$push`：向字段追加值
> - `$pushAll`：向字段追加一组值
> - `$addToSet`：将值添加到数组中，重复的不添加
> - `$pop`：删除数组末尾元素
> - `$pull`：给定一个值，删除字段中所有匹配元素
> - `$pullAll`：给定一组值，删除字段中所有匹配元素
> - `$rename`：重命名字段

缺失隔离级别有时会导致脏读。数据被修改后，游标不会自动更新。

MongoDB 默认每分钟刷一次磁盘，到时候插入和更新的数据就会记录到磁盘上。两次更新之间，任何错误都可能导致不一致。可以提高同步频率或强制刷磁盘，但这样会影响性能。

为了避免在系统错误时丢失所有更新，明智的做法是进行复制。可以按主从结构配置两个 MongoDB 实例进行复制和数据同步。复制是异步的，因此修改不会即刻完成复制，不过有总比没有强。目前的 MongoDB 版本中，副本集代替了主从复制，一个副本集包括三个副本。一架长机，两架僚机。副本集支持自动恢复和自动故障转移。

复制更多的是作为一种故障转移和灾难恢复准备，而分片则可用来进行水平扩展。

4.3.4　水平扩展

选用 MongoDB 的一个常见原因和弱 schema 集合有关，另外一个原因则是其良好的性能和扩展性。最近的版本中，MongoDB 开始支持自动分片，这使得水平扩展更容易了。

分片的基本概念和列数据库的主从模式非常相近，主从模式把数据分散到多台 range server 上。MongoDB 支持把集合分开保存到多台机器上，每台机器保存一部分，算一个分片。故障转移通过复制分片实现。例如，一个大集合可以分成四个分片，每个分片复制三份，总共创建 12 个 MongoDB 服务器单元。每个分片多出来的两份拷贝用作故障转移单元。

分片是集合层面的，不是数据库层面的。因此，可能同一个数据库里的集合 A 驻留在单个节点上，而集合 B 则分片到多个节点上。

每个分片存储文档有序集合中连续的一部分。这一部分按 MongoDB 的话说就是块。每个块由三个属性来标识：首文档键（最小键）、尾文档键（最大键）、集合。

任何有效的片键模式都可以用来对集合分片。任何一个字段、两个或更多文档字段的组合都可以用作片键。除字段以外，片键还包括排序方向的属性。排序方向 1 表示升序，–1 表示降序。

分片选择非常重要，需要审慎且有远见，以保证划分后的平衡性。

所有与分片及块有关的定义都保存在配置服务器的元数据目录里。和分片一样，配置服务器也通过复制支持故障转移。

客户端进程通过 mongos 连接 MongoDB 集群。mongos 进程本身不保存状态，而是从配置服务器中获取。一个 MongoDB 集群可以有一个或多个 mongos 进程。mongos 进程负责请求路由和结果合并。发送给 MongoDB 集群的查询可以是有针对性的，也可以是全局的。可以利用片键（数据通常按片键排序）的查询是针对性查询，反之就是全局的。针对性查询比全局查询效率更好。可以把全局查询看作需要全集合扫描的查询。

MongoDB 分片架构的拓扑如图 4-11 所示。

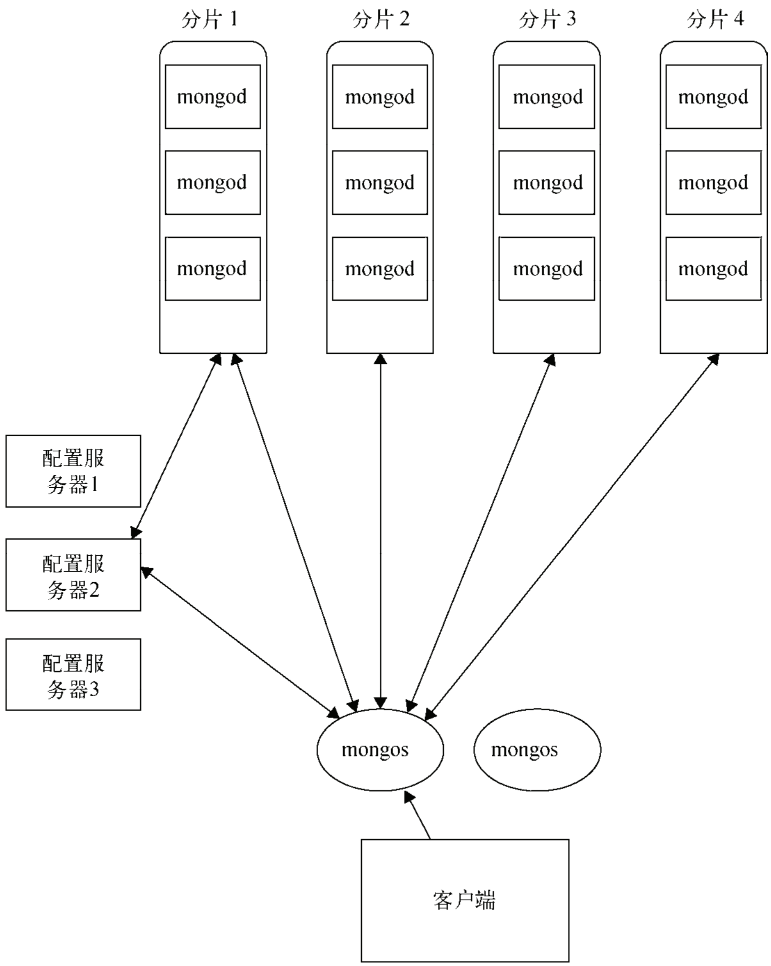


图 4-11

接下来我们将介绍键/值存储的存储架构及其彼此间的微小差别。

4.4　键/值存储 Memcached 和 Redis

虽然各种键/值存储千差万别，但是它们也有很多共通之处，比如数据都存储在映射表里。本节将展示 Memcached 和 Redis 的内部机制，介绍一个健壮的键/值存储都由哪些部分组成。

4.4.1　Memcached的内部结构

Memcached（下载请访问 http://memcached.org）是一种分布式高性能对象缓存系统，它风靡一时，普遍应用于高流量网站，例如 Facebook、Twitter、Wikipedia 和 Youtube。Memcached 简洁到了极致，只包含最小的功能集，不支持备份、故障转移或者故障恢复。它提供的 API 很简单，几乎任何 Web 编程语言都可以使用。应用程序栈中使用 Memcached 的主要目的通常是减少数据库负载。典型 Web 应用中 Memcached 可能的配置见图 4-12。

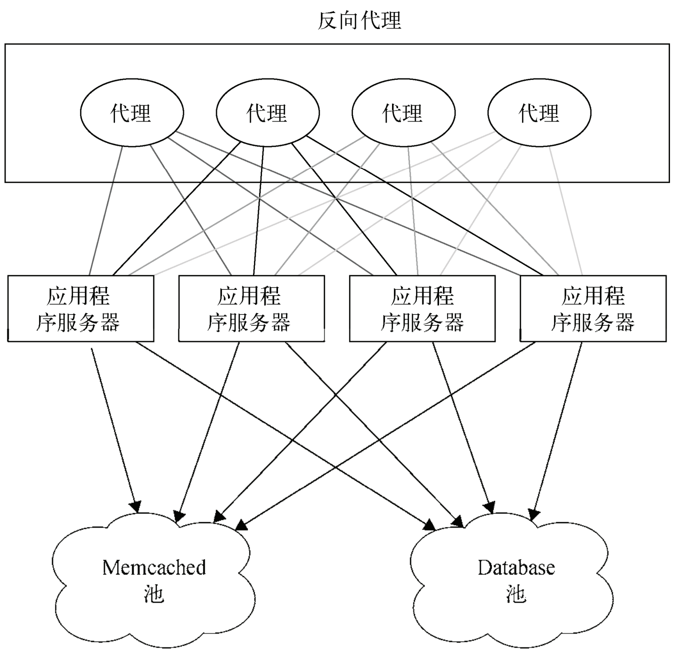


图　4-12

Memcached 的核心是一个槽（slab）分配器。Memcached 按槽存储值。槽本身由页（page）组成，页又由块（chunk）或桶（bucket）组成。槽最小 1kB，大小按 1.25 的幂次增长。因此槽的大小可以是 1kB（1.25 的 0 次幂）、1.25kB（1.25 的 1 次幂）、1.5625（1.25 的 2 次幂）等，以此类推。Memcached 可以存储的值最大不能超过 1MB。值通过键来存储和引用。键最大是 250

字节。对象存储在与其大小最近似的块或桶中。例如一个 1.4kB 大小的对象，就会被存储在 1.5625kB 大小的块中。这会导致空间浪费，特别是当对象仅略大于上一个比它小的块时。默认情况下，Memcached 能用掉所有可用的内存，它只受限于底层架构。Memcache 的一些基本特征如图 4-13 所示。

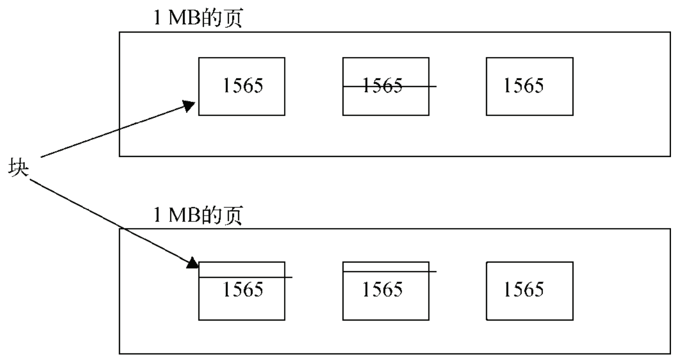


图 4-13

LRU 算法控制对旧缓存对象的驱逐，以槽为单位执行。随着对象不断被存储、清理，有可能会出现碎片，而重新分配内存能解决部分问题。

Memcached 作为一种对象缓存，组织数据元素时并不使用某种集合的形式，比如列表、无序集、有序集或映射表。但是 Redis 提供了对所有这些丰富的数据结构的支持。Redis 类似于 Memcached，但更健壮。前面章节中我们使用过了 Redis。

下面简要地介绍一下 Redis 的内部结构。

4.4.2 Redis的内部结构

在 Redis 中万物皆为字符串，甚至像列表、无序集、有序集和映射表这样的集合也由字符串组成。Redis 定义了一个特别的结构 SDS，称为简单动态字符串（simple dynamic string），这个结构由以下三部分组成。

- **`buff`**：存储字符串的字符数组
- **`len`**：存储 buff 数组长度的长整型
- **`free`**：可用字节的数量

单独存储 `len` 看似是多余的开销，因为通过 buff 数组可以很容易计算出来，但是它能在固定时间内返回字符串长度。

Redis 在主内存中保存数据，并按需将其持久化到磁盘上。与 MongoDB 不同，它没有使用内存映射文件。相反，Redis 实现了它自己的虚拟内存子系统。当一个值被换出到磁盘上时，一个指向那个磁盘页的指针会和键一起存储。更多有关其虚拟内存技术规范的内容，请访问：http://code.google.com/p/redis/wiki/VirtualMemorySpecification。

除了虚拟内存管理器以外，Redis 还包括了一个事件库，用来协调非阻塞的套接字操作。Redis 架构概览如图 4-14 所示。

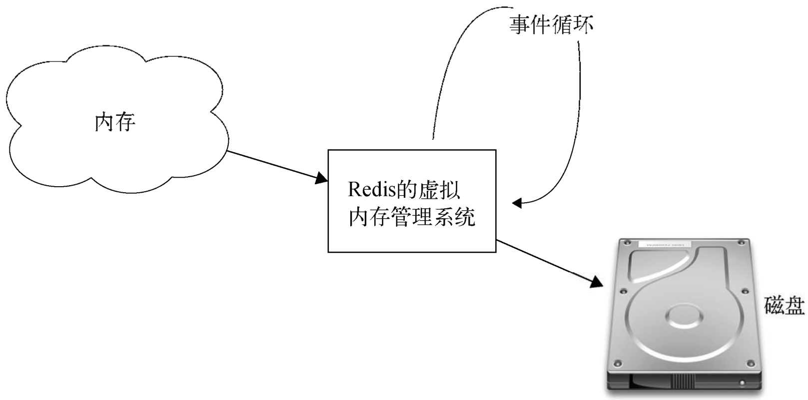


图　4-14

为什么 Redis 不依赖操作系统的虚拟内存交换？

Redis 不依赖操作系统交换，是出于以下原因。

- Redis 对象并不与内存页一一映射。一页是 4096 字节长，而一个 Redis 对象可以跨越多个页。类似地，多个 Redis 对象也可以放在一页上。因此，即使只访问少量 Redis 对象，也可能触及大量页。操作系统会跟踪对页的访问。因此，即使一页里只有一字节被访问过，它也会被排除到交换系统之外。
- 与 MongoDB 不同，Redis 数据在 RAM 里和在磁盘上格式不一样。在磁盘上的数据经过压缩远小于在 RAM 里的大小，因此使用自定义的交换技术能减少磁盘 I/O。

Redis 的作者 Salvatore Sanfillipo 在他的博文"Redis Virtual Memory: the story and the code"中讨论了 Redis 的虚拟内存系统，查阅这篇博文请访问：http://antirez.com/post/redis-virtual-memory-story.html。

接下来，也是最后，我们把注意力转移回面向列的数据库。不过这一次是一族特殊的面向列数据库，即那些最终一致的面向列数据库。

4.5　最终一致性非关系型数据库

如果说 Google Bigtable 是列数据库的灵感之源，那么 Amazon Dynamo 就是最终一致性存储的原型。2007 年，在会议 Symposium on Operating Systems Principles 上，Amazon Dynamo 背后的理念被提了出来，并通过一篇技术论文开放给了公众，论文地址是：www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf。没过多久，Dynamo 的思想就融入到了许多开源实现中，例如

Apache Cassandra、Voldemort、Riak 和 Dynomite。本节将讨论最终一致性键/值存储的基本原理，具体的开源实现留待后面章节再讨论。

Amazon Dynamo 支撑了亚马逊内部许多服务的运转，这些服务又控制着它庞大的电子商务系统。这个系统有些基本的需求，包括高可用性和容错能力。不过数据被组织成了大部分情况下只需按主键查询的结构，所以关系型引用和 join 连接不是必需的。Dynamo 建立在一致性哈希（consistent hashing）、对象版本（object versioning）、闲话协议（gossip-based membership protocol）、哈希树（merkle tree，又叫 hash tree）和提示移交（hinted handoff）这些想法的基础上。

Dynamo 支持简单的 get-put 接口。Put 请求包含与对象版本有关的数据，这个数据存储在上下文里。Dynamo 支持随数据增长扩容，因此，它依赖一致性哈希进行有效的分区。

4.5.1 一致性哈希

一致性哈希已经成为分布式哈希表一个重要的原则。一致性哈希中添加或者删除一个槽（slot）不会显著改变键与槽的映射关系。要明白这种哈希方案的价值，我们先来看看基本的哈希方案，理解在增删槽时会出现什么问题。

一个简朴的键分配策略是取余函数。要把 50 个键分配给 7 个节点，可以这样做：键为 85 的去节点 1，因为 85 对 7 取余等于 1；键为 18 的去节点 4，因为 18 对 7 取余得 4；其他的以此类推。只要节点数量不发生变化（即添加新节点或者删除老节点），这种策略就没什么问题。一旦节点数量发生变化，对原有的键取余函数会产生跟先前不同的输出，导致键在节点的位置重新排列。出现这种低效的情况时，一致性哈希就有了用武之地。

一致性哈希中，节点的增减不会对键的排列产生非常大的影响。为了直观起见，我们画一个圆来解释，如图 4-15，上面标记出了节点的位置。

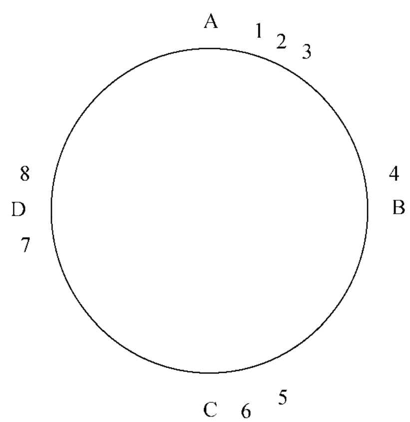


图 4-15

键应该分配给离它们最近的节点。图 4-15 里，1、2、3 分配给节点 A；4 分配给 B；5 和 6 分配给 C；7 和 8 给 D。这种方案需要很大的哈希空间，通常包含 SHA1 键的所有空间，然后将

其映射到一个圆上。从 0 开始，沿顺时针方向依次映射直到最大值为止，然后回到 0。节点也用相同的办法哈希和映射。

现在，假设移除节点 A，添加节点 E，如图 4-16。然后重排，1 去节点 E，2、3 去节点 B，其他值都不受影响。

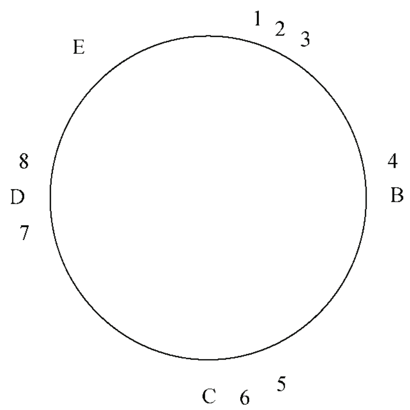


图　4-16

一致性哈希提供了有效的分区，而对象版本则有助于保持数据一致。

4.5.2　对象版本

对大型的分布式和高可扩展系统，ACID 事务会带来巨大的开销，所以 Dynamo 提出了用对象版本和矢量时钟来保持一致性。下面举一个例子来讲讲矢量时钟是如何工作的。

假设有四个黑客：Joe、Hillary、Eric 和 Ajay，他们决定见面聊聊矢量时钟。Joe 建议在 Palo Alto 碰头。之后，Hillary 和 Eric 在工作时见面，并认为 Mountain View 是最佳的聚会地点。同一天，Eric 和 Ajay 相互通了消息，并得出结论：应该在 Los Altos 碰头。等到了聚会的那天，Joe 给每个人发了带提醒的电子邮件和 Palo Alto 的聚会地址。Hillary 回应说会场改在了 Mountain View，而 Ajay 说应该是 Los Altos。两个人都声称 Eric 知道决定。现在大家联系 Eric 来解决这个问题。这时可以用矢量时钟来解决冲突。

我们可以为三个会场 Palo Alto、Mountain View 和 Los Altos 分别创建矢量时钟如下：

```
Venue: Palo Alto
Vector Clock: Joe (ver 1)

Venue: Mountain View
Vector Clock: Joe (ver 1), Hillary (ver 1), Eric (ver 1)

Venue: Los Altos
Vector Clock: Joe (ver 1), Ajay (ver 1), Eric (ver 1)
```

Mountain View 和 Los Altos 的矢量时钟都包括了 Joe 最初的选择，因为大家都知道。Moutain

View 的矢量时钟是基于 Hillary 的回应，而 Los Altos 的矢量时钟是基于 Ajay 的回应。Mountain View 和 Los Altos 的矢量时钟没有同步，因为它们不是从对方那里延伸（descend）下来的。只有当一个矢量时钟的版本大于等于另一个矢量时钟的所有值时，才能算是从它那里延伸下来的。

最后，Joe 在电话上抓着 Eric 不放，要他解决混乱的状态。Eric 意识到了问题，并快速得出结论：在 Mountain View 举行会议可能是最好的想法。现在 Joe 画出更新后的矢量时钟如下：

```
Venue: Palo Alto
Vector Clock: Joe (ver 1)

Venue: Mountain View
Vector Clock: Joe (ver 1), Hillary (ver 1), Ajay (ver 0), Eric (ver 2)

Venue: Los Altos
Vector Clock: Joe (ver 1), Hillary (ver 0), Ajay (ver 1), Eric (ver 1)
```

版本 0 是为 Hillary 和 Ajay 创建的，代表他们没有提议过，但是现在知道了的会场。现在，矢量时钟都相互延伸下来了，而且确定 Mountain View 是他们碰面的地点。从这个例子中可以观察到，矢量时钟不仅能帮助确定事件发生的顺序，还能标识出产生这些问题的根源，进而帮助解决不一致性。

除了对象版本以外，Dynamo 还在节点上使用闲话协议，并用提示移交来保证一致性。

4.5.3 闲话协议和提示移交

闲话协议是一种通讯协议，它的灵感来自社交网络上和办公室里的八卦。闲话协议涉及周期、配对以及跨进程通讯。它的可靠性不高，而且配对往往是随机的。

为了持久性，消息通常要求写入所有指定节点，而提示移交放宽了这个要求。在健康节点上执行了写操作就行，同时要添加一个提示，以便失败节点重启后能得到消息。

4.6 小结

本章简要介绍了 NoSQL 数据库的基本概念；解释了有关流行 NoSQL 数据库的数据模型、存储方案和配置的一些基础知识；展示了面向列数据库的典型构成，用 HBase 演示了一些共通的主题；讲解了文档数据库和键/值存储的内部结构；最后介绍了最终一致性数据库。

第 5 章 执行 CRUD 操作

本章内容

- 介绍 NoSQL 数据库的创建、读取、更新和删除操作
- 解释为什么重视创建胜过更新
- 研究更新操作的原子性和完整性
- 解释持久化相关数据的各种方式

基本操作（创建、读取、更新和删除，即 CRUD，俗称为增删改查）是数据交互最基本的方式。所以了解如何在 NoSQL 世界中应用这些操作非常重要。NoSQL 并不是一个产品或一项技术，而是一类数据库的总称，因此 CRUD 操作的含义随 NoSQL 产品不同而变化。不过它们存在这样一个共有的突出特征：在 NoSQL 存储中，创建和读取操作比更新和删除操作更重要，以至于有时只存在创建和读取操作。在接下来的几节中，你就能了解其中的含义。当我们从 CRUD 操作的角度逐渐深入探索 NoSQL 新大陆时，按照逻辑，相关的内容会被划分为三部分：面向列的、以文档为中心的以及键/值映射表。

CRUD 中第一根支柱是创建操作。

5.1 创建记录

创建记录操作几乎不用定义。第一次保存一条新记录时，就会创建一个新条目。这意味着应该有某种方法来轻松标识一条记录，以及确定它是否已经存在。如果存在，可能要更新记录，而不是重新创建记录。

关系型数据库中，记录存储在表中，主键唯一标识所有记录。要检查一条记录是否存在，只要检查记录的主键是否在表里就行。如果一条记录没有主键，但是所有列或字段的值都与已存在的一条记录完全匹配，又会怎样？这正是事情开始变得棘手的地方。

关系型数据库维护着由 E.F. Codd 引入的规范化原则，它是关系模型的一部分。E.F. Codd 和 Raymond F. Boyce 在 1974 年提出了 BC 范式（BCNF，Boyce-Codd Normal Form），它常被看作保持数据库结构规范化的最低要求。不严格地说，符合范式的结构能帮助减少对记录集的修改，这是通过只保存一次数据，在需要的时候才创建对相关数据的引用做到的。要了解更多有关数据库规范化的内容，请访问 http://en.wikipedia.org/wiki/Database_normalization 和 http://databases.about.

com/od/specificproducts/a/normalization.htm。

在规范化结构中，两个具有相同值的记录就是相同的记录。因此实际上关系模型的主键（单一列）中隐含着按值比较的规则。在编程语言的世界里，特别是在面向对象语言中，这个概念通常被替换成按引用比较，每个唯一记录集以一个对象的形式存在，对象用内存地址唯一标识。因为 NoSQL 数据库类似表格结构和对象存储，所以标识的语义从基于值的到基于引用的都有。但是无论哪种情况，唯一主键标识的概念都非常重要，它能用来确定一条记录。

大多数据库都支持使用任意字符串或字节数组作为唯一记录键，除此外，它们往往还会预定义一些规则以确保键唯一且有意义。而在一些数据库里，可以借助工具函数来生成主键。

唯一主键

前面见过 MongoDB 的默认 BSON 对象标识（参见前一章图 4-10），它是由 12 个字节构成一个键，结构如下：

- 开始四个字节表示时间戳
- 跟着三个字节表示机器标识符
- 跟着二个字节表示进程标识符
- 最后三个字节是顺序或者递增计数器

我们还见过 HBase 行键，通常由字符表示的字节数组组成。HBase 行键通常有 64 字节长，不过这并不成为限制，尽管更大的键会占用更多内存。HBase 的数据按行键的字节排序，所以定义与应用程序有关的逻辑单元作为行键会非常有用。

明白了记录标识符，接下来介绍在 NoSQL 数据库中如何创建记录。前面几章中，我们分别使用 MongoDB、HBase 和 Redis 作为以文档为中心的、面向列的和键/值映射表的例子。在本节中，会再次利用这三种数据库。

5.1.1　在以文档为中心的数据库中创建记录

很多关系型数据库都使用过一个典型例子，这个例子是一个简化的零售系统，用来创建和管理订单记录。每个人在商店里的购买都形成订单，一个订单由好几个条目组成，每个条目包括产品（条目）和购买的数量。条目还包括价格，价格是通过计算产品单价和个数的乘积得到的。订单表有一个相关联的产品表，该表存储产品描述和其他产品相关属性。图 5-1 用传统的实体关系图描述了订单、产品和它们之间的关系表。

对于像 MongoDB 这样的文档存储，要保存同样的数据，需要对结构进行去规范化，把每个条目详情和订单记录本身一起存储。举一个具体的例子，比如有一个包含四种咖啡的订单：拿铁 1 份，卡布其诺 1 份，普通咖啡 2 份。这个咖啡订单会以嵌套的类 JSON 文档形式存储在 MongoDB 中，如下所示：

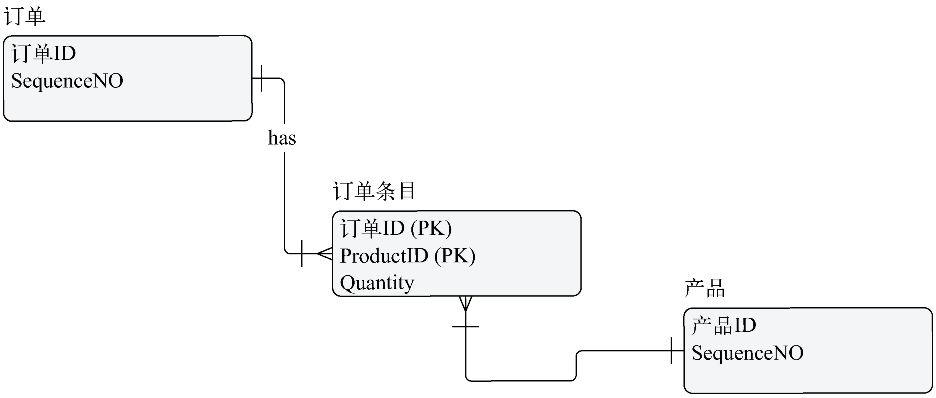


图　5-1

```
{
    order_date: new Date(),
    "line_items": [
        {
            item : {
                name: "latte",
                unit_price: 4.00
            },
            quantity: 1
        },
        {
            item: {
                name: "cappuccino",
                unit_price: 4.25
            },
            quantity: 1
        },
        {
            item: {
                name: "regular",
                unit_price: 2.00
            },
            quantity: 2
        }
    ]
}
```

coffee_order.txt

打开命令行窗口，转到 MongoDB 的根目录下面，启动 MongoDB 服务器：

```
bin/mongod --dbpath ~/data/db
```

现在，在另外一个窗口中启动命令行客户端来与服务器进行交互：

```
bin/mongo
```

用命令行客户端存储咖啡订单，并保存在 mydb 数据库的 orders 集合里。命令行中部分命令的输入和输出列举如下：

```
> t = {
...     order_date: new Date(),
...     "line_items": [ ...
...     ]
... };

{
    "order_date" : "Sat Oct 30 2010 22:30:12 GMT-0700 (PDT)",
    "line_items" : [
        {
            "item" : {
                "name" : "latte",
                "unit_price" : 4
            },
            "quantity" : 1
        },
        {
            "item" : {
                "name" : "cappuccino",
                "unit_price" : 4.25
            },
            "quantity" : 1
        },
        {

            "item" : {
                "name" : "regular",
                "unit_price" : 2
            },
            "quantity" : 2
        }
    ]
}

> db.orders.save(t);
> db.orders.find();
{ "_id" : ObjectId("4cccff35d3c7ab3d1941b103"), "order_date" : "Sat Oct 30 2010
22:30:12 GMT-0700 (PDT)", "line_items" : [
    ...
] }
```

coffee_order.txt

尽管建议存储整个嵌套文档，但有时也需要分开存储嵌套对象。嵌套对象分开存储以后，将记录集合并起来的责任就落到了开发者的身上。MongoDB 中没有数据库 join 的概念，因此要么在客户端利用对象标识符手工实现 join 操作，要么利用 DBRef。

> DBRef 是在 MongoDB 里用于创建文档间引用的正式规范。DBRef 包括集合名和对象标识符。更多关于 MongoDB DBRef 的信息请访问：www.mongodb.org/display/DOCS/Database+References#DatabaseReferences-DBRef。

现在我们调整一下结构，不把产品单价存储在嵌套文档中，而是保存在另外一个集合里，这个集合专门存储产品信息。在新格式中，产品名是连接两个集合的关键。

这样，重构后的订单数据存储在 orders2 集合中，如下所示：

```
> t2 = {
...     order_date: new Date(),
...     "line_items": [
...         {
...             "item_name":"latte",
...             "quantity":1
...         },
...         {
...             "item_name":"cappuccino",
...             "quantity":1
...         },
...         {
...             "item_name":"regular",
...             "quantity":2
...         }
...     ]
... };
{
    "order_date" : "Sat Oct 30 2010 23:03:31 GMT-0700 (PDT)",
    "line_items" : [
        {
            "item_name" : "latte",
            "quantity" : 1
        },
        {
            "item_name" : "cappuccino",
            "quantity" : 1
        },
        {
            "item_name" : "regular",
            "quantity" : 2
        }
    ]
}
> db.orders2.save(t2);
```

coffee_order.txt

要确认数据被正确地存储，可以像下面这样返回集合 orders2 的内容：

```
> db.orders2.find();
{ "_id" : ObjectId("4ccd06e8d3c7ab3d1941b104"), "order_date" : "Sat Oct 30 2010
 23:03:31 GMT-0700 (PDT)", "line_items" : [
    {
        "item_name" : "latte",
        "quantity" : 1
    },
...
] }
```

coffee_order.txt

下面，保存产品数据，其中包括产品名和单价，如下所示：

```
> p1 = {
...     "_id": "latte",
...     "unit_price":4
... };
{ "_id" : "latte", "unit_price" : 4 }
> db.products.save(p1);
```

coffee_order.txt

接着可以使用 find 方法验证 products 集合中的记录：

```
> db.products.find();
{ "_id" : "latte", "unit_price" : 4 }
```

coffee_order.txt

现在就可以手工连接两个集合并取出相关数据集，如下所示：

```
> order1 = db.orders2.findOne();
{
    "_id" : ObjectId("4ccd06e8d3c7ab3d1941b104"),
    "order_date" : "Sat Oct 30 2010 23:03:31 GMT-0700 (PDT)",
    "line_items" : [
        {
            "item_name" : "latte",
            "quantity" : 1
        },
        {
            "item_name" : "cappuccino",
            "quantity" : 1
        },
        {
            "item_name" : "regular",
            "quantity" : 2
        }
    ]
}
> db.products.findOne( { _id: order1.line_items[0].item_name } );
{ "_id" : "latte", "unit_price" : 4 }
```

coffee_order.txt

除了手工处理，还可以借助 DBRef 来自动完成这个手工过程的一部分，DBRef 是一个更正式的用来关联 MongoDB 两个文档集合的规范。要演示 DBRef，得重新调整订单例子，建立关系。首先定义产品，然后设置一个从 orders 集合指向产品的 DBRef。

添加 latte、cappuccino、regular 及其对应的单价到 product2 集合如下：

```
> p4 = {"name":"latte", "unit_price":4};
{ "name" : "latte", "unit_price" : 4 }
> p5 = {
...     "name": "cappuccino",
...     "unit_price":4.25
... };
{ "_id" : "cappuccino", "unit_price" : 4.25 }
> p6 = {
...     "name": "regular",
...     "unit_price":2
... };
{ "_id" : "regular", "unit_price" : 2 }
> db.products2.save(p4);
> db.products2.save(p5);
> db.products2.save(p6);
```

coffee_order.txt

确认所有三个产品都在集合中：

```
> db.products.find();
{ "_id" : ObjectId("4ccd1209d3c7ab3d1941b105"), "name" : "latte",
  "unit_price" : 4 }
{ "_id" : ObjectId("4ccd1373d3c7ab3d1941b106"), "name" : "cappuccino",
  "unit_price" : 4.25 }
{ "_id" : ObjectId("4ccd1377d3c7ab3d1941b107"), "name" : "regular",
  "unit_price" : 2 }
```

coffee_order.txt

接下来，定义一个新的 orders 集合，叫做 orders3，并用 DBRef 来建立 order3 和 products 之间的关系。集合 order3 可以定义如下：

```
t3 = {
...     order_date: new Date(),
...     "line_items": [
...         {
...             "item_name": new DBRef('products2', p4._id),
...             "quantity:1
...         },
...         {
...             "item_name": new DBRef('products2', p5._id),
...             "quantity":1
...         },
...         {
...             "item_name": new DBRef('products2', p6._id),
...             "quantity":2
```

```
...          }
...      ]
... };

db.orders3.save(t3);
```

coffee_order.txt

在 MongoDB 中创建记录的过程非常简单，有些关系可以用 DBRef 建立。接下来，我们来了解面向列数据库的创建操作。

5.1.2 面向列数据库的创建操作

与 MongoDB 不同，面向列数据库没有定义任何关系引用的概念。和所有 NoSQL 产品一样，它们也避免了集合之间的连接，所以不存在外键或跨集合约束之类的概念。列数据库采用去规范化的方式存储集合，几乎类似于一个存储大量去规范化事务记录的数据仓库事实表。列数据库里数据是这样存储的：每个行键唯一标识一条记录，一个列族的所有列存储在一起。

面向列的数据库，特别是 HBase，也有一个保存数据的时间维度。因此创建（数据插入）操作非常重要，而更新操作的概念基本上就不存在了。举个例子来看看。假设需要创建和维护一个巨大的目录，包含不同类型的产品信息，其中关于产品类型、类别、特征、价格和来源的信息可以有很大的变化。我们要创建一个表，包含类型、特征和来源三个列族。每个属性或字段（或者说列）都从属于这些列族中的一个。要在 HBase 中创建这个集合（产品表），首先启动 HBase 服务器，然后用 HBase shell 连上服务器。要启动 HBase 服务器，打开一个命令行窗口或终端，并调整使其进入 HBase 安装目录，接着以本地模式启动 HBase 服务器，操作如下：

```
bin/start-hbase.sh
```

启动另一个命令行窗口并用 HBase shell 连上服务器：

```
bin/hbase shell
```

接下来，创建 product 表：

```
hbase(main):001:0> create 'products', 'type', 'characteristics', 'source'
0 row(s) in 1.1570 seconds
```

products_hbase.txt

创建好表以后，就可以存数据了。HBase 使用 put 关键字表示数据创建操作。单词“put”说明数据插入是类似哈希映射表的操作，又加上 Hbase 内部很像一个嵌套的哈希表，因此它可能比 create 关键字更合适。

要创建一个包含下列字段的记录：

- type:category = "coffee beans"
- type:name = "arabica"
- type:genus = "Coffea"
- characteristics: cultivation_method = "organic"

- characteristics: acidity = "low"
- source: country = "yemen"
- source: terrain = "mountainous"

可以像下面这样把它插入 products 表中：

```
hbase(main):001:0> put 'products', 'product1', 'type:category', 'coffee beans'
0 row(s) in 0.0710 seconds
hbase(main):002:0> put 'products', 'product1', 'type:name', 'arabica'
0 row(s) in 0.0020 seconds
hbase(main):003:0> put 'products', 'product1', 'type:genus', 'Coffea'
0 row(s) in 0.0050 seconds
hbase(main):004:0> put 'products', 'product1',
  'characteristics: cultivation_method', 'organic'
0 row(s) in 0.0060 seconds
hbase(main):005:0> put 'products', 'product1', 'characteristics: acidity', 'low'
0 row(s) in 0.0030 seconds
hbase(main):006:0> put 'products', 'product1', 'source: country', 'yemen'
0 row(s) in 0.0050 seconds
hbase(main):007:0> put 'products', 'product1', 'source: terrain', 'mountainous'
0 row(s) in 0.0050 seconds
hbase(main):008:0>
```

products_hbase.txt

现在查询同一条记录以确保它已经被放入存储中。获取记录的操作如下：

```
hbase(main):008:0> get 'products', 'product1'
COLUMN                          CELL
characteristics: acidity        timestamp=1288555025970, value=lo
characteristics: cultivatio timestamp=1288554998029, value=organic
n_method
source: country                 timestamp=1288555050543, value=yemen
source: terrain                 timestamp=1288555088136, value=mountainous
type:category                   timestamp=1288554892522, value=coffee beans
type:genus                      timestamp=1288554961942, value=Coffea
type:name                       timestamp=1288554934169, value=Arabica
7 row(s) in 0.0190 seconds
```

products_hbase.txt

如果再存一次 type:category 的值，这次用 beans 代替 coffee beans，会怎样呢？

```
hbase(main):009:0> put 'products', 'product1', 'type:category', 'beans'
0 row(s) in 0.0050 seconds
```

products_hbase.txt

现在，如果再次获取记录，那么输出如下：

```
hbase(main):010:0> get 'products', 'product1'
COLUMN                          CELL
characteristics: acidity        timestamp=1288555025970, value=low
characteristics: cultivatio timestamp=1288554998029, value=organic
n_method
source: country                 timestamp=1288555050543, value=yemen
source: terrain                 timestamp=1288555088136, value=mountainous
```

```
type:category            timestamp=1288555272656, value=beans
type:genus               timestamp=1288554961942, value=Coffea
type:name                timestamp=1288554934169, value=Arabica
7 row(s) in 0.0370 seconds
```

products_bbase.txt

现在 `type:category` 的值是 `beans` 而非 `coffee beans`。但是实际上，两个值都是作为同一个字段值的不同版本被保存起来，只是默认会返回其中的最新值。要查看 `type:category` 字段最近的四个版本，执行下面的命令：

```
hbase(main):011:0> get 'products', 'product1', { COLUMN => 'type:category',
  VERSIONS => 4 }
COLUMN                        CELL
type:category                timestamp=1288555272656, value=beans
type:category                timestamp=1288554892522, value=coffee beans
```

目前只有两个版本，所以就只返回了这些。

如果数据是结构化的、有限的，而且天生就是关系型的，那么有可能 HBase 并不是正确的解决方案。

HBase 使数据的结构扁平化，只有列族和它包含的列之间才有层次关系。此外，它还按时间维度存储每个单元格的数据，所以将数据存储入 HBase 前，需要对嵌套的数据进行扁平化处理。

下面来分析零售订单系统的问题。在 HBase 中，零售订单数据可以用以下多种方式存储。

- 扁平化所有数据，把一个订单的所有字段（包括所有产品数据）都存储到一条记录中。
- 每个订单的所有订单条目存储在一条记录中。产品信息单独放一个表，对产品行键的引用和订单条目信息保存在一起。

如果选第一种方案，就要做出下面这些决定。

- 为普通的条目创建一个列族，然后为其他类型的条目（比如打折或回扣）创建另一个列族。
- 在一个普通条目的列族里，可以有这些列：条目或产品名、条目或产品描述、数量和价格。如果全部数据都要扁平化，记得每个条目都要分配不同的键，不然它们最终会变成同一键/值对的不同版本。比如产品名应该叫 `product_name_1`，千万别都叫 `product_name`。

下一个例子将使用 Redis 来阐示在键/值映射表中创建数据。

5.1.3　键/值映射表的创建操作

Redis 是一种简单强大的数据结构服务器，支持以简单的键/值对或集合元素的形式来存储数据。每个键/值对可以是独立的字符串映射表，也可以放在集合里。一个集合可以是以下任何类型：列表（list）、无序表（set）、有序表（sorted set）或哈希表（hash）。单独的字符串键/值对很像一个可以接受字符串值的变量。

创建一个 Redis 字符串键/值映射表：

```
./redis-cli set akey avalue
```

可以用 get 命令来确认创建成功：

```
./redis-cli get akey
```

响应（正如所料）是 avalue。set 方法和 create 方法（或者 put 方法）相同。如果再次调用 set 方法，并对 akey 使用值 anothervalue，原来的值会被新值替换。试着输入：

```
./redis-cli set akey anothervalue
./redis-cli get akey
```

结果肯定是新值 anothervalue。

字符串方法 set 和 get 不能用于 Redis 集合。例如，用 lpush 和 rpush 来创建和填充一个列表，一个尚不存在的列表可以和它的第一个成员一起被创建，像这样：

```
./redis-cli lpush list_of_books 'MongoDB: The Definitive Guide'
```

books_list_redis.txt

可以使用范围操作来确认和查看列表 list_of_books 的前面几个成员，像这样：

```
./redis-cli lrange list_of_books  0 -1
1. "MongoDB: The Definitive Guide"
```

books_list_redis.txt

范围操作使用第一个元素的索引 0 和最后一个元素的索引-1 来获取列表中的所有元素。

在 Redis 中，如果查询一个不存在的列表，它会返回一个空列表，不会抛出异常。可以像下面这样对一个不存在的列表 mylist 执行范围操作：

```
./redis-cli lrange mylist 0 -1
```

Redis 会返回消息：empty list or set。可以像使用 lpush 那样用 rpush 添加一个新成员到 mylist 中，像这样：

```
./redis-cli rpush mylist 'a member'
```

现在 mylist 肯定不是空的了，使用范围查询可以揭示出 a member 的存在。

成员可以从列表的左侧或右侧插入列表中，也可以从任一方向弹出。这样就可以把列表当作队列或栈来使用。

无序集可以用 SADD 操作添加成员。因此，可以用下面的命令添加成员'a set member'到无序集 aset 中：

```
./redis-cli sadd aset 'a set member'
```

命令行程序返回整数值 1，确认'a set member'已经被添加进无序集。再次执行同样的 SADD 命令，成员不会被重复添加。无序集对同一个值仅保存一份，因此一旦已经存在就不会再添加了。注意程序返回 0 作为响应，这意味着没有添加成员。与无序集类似，有序集中每个成员也只保存一份，不过有序集像列表一样有顺序。可以轻松添加成员'a sset member'到有序集

azset 中，像这样：

```
./redis-cli zadd azset 1 'a sset member'
```

1 代表有序集成员的位置（或分值）。下面再添加另外一个成员'sset member 2'到同一个有序集中：

```
./redis-cli zadd azset 4 'sset member 2'
```

运行类似于列表的范围操作可以帮助确认这些值已被存储。有序集的范围指令是 zrange，可以像下面这样获取包含前五个值的范围：

```
./redis-cli zrange azset 0 4
1. "a sset member"
2. "sset member 2"
```

如果添加一个位置（或分值）为 3 的值会怎样呢？在已经存在值的位置（或分值）4 上添加呢？

添加一个分值为 3 的值到 azset 中，如下：

```
./redis-cli zadd azset 3 'member 3'
```

接着运行 zrange 查询如下：

```
./redis-cli zrange azset 0 4
```

结果：

```
1. "a sset member"
2. "member 3"
3. "sset member 2"
```

再添加位置（或分值）为 3 的值，像这样：

```
./redis-cli zadd azset 3 'member 3 again'
```

然后运行 zrange 查询：

```
./redis-cli zrange azset 0 4
```

可以看出，为适应新成员，有序集中的成员被重新定位：

```
1. "a sset member"
2. "member 3"
3. "member 3 again"
4. "sset member 2"
```

因此，添加新成员到有序集中不会替换掉已经存在的值，而是会按需对成员重新排序。

Redis 还定义了一个哈希的概念，其中可以这样添加成员：

```
./redis-cli hset bank account1 2350
./redis-cli hset bank account2 4300
```

可以用 hget 命令或其变种 hgetall 确认成员存在，输入：

```
./redis-cli hgetall bank
```

要存储一个复杂的嵌套哈希，可以像下面这样创建一个层级哈希键：

```
./redis-cli hset product:fruits apple 1.35
./redis-cli hset product:fruits banana 2.20
```

数据被保存到任何一种 NoSQL 数据存储中后，就需要去访问和获取它们。毕竟，保存数据的意义就是以后还能再拿出来用。

5.2　访问数据

前面我们见过一些访问数据的方法。为了验证记录已被创建，使用过一些最简单的 get 命令。较早的章节里也演示了一些标准的查询机制。

接下来学习几种高阶数据访问方法、语法和语义。

5.2.1　用MongoDB访问文档

MongoDB 的文档查询使用与 SQL 非常类似的语法和语义。比较讽刺的是，这种 SQL 相似性使得 NoSQL 的 MongoDB 查询文档非常方便和强大。

前面章节里我们已经熟悉了查询文档的使用，所以可以直接开始访问 MongoDB 嵌套文档。这次我们还是用前面章节里创建好的 mydb 数据库的 orders 集合。

启动 MongoDB 服务器，用 mongo JavaScript shell 连上它。执行 use mydb 命令指向 mydb 数据库。首先，获取 orders 集合中的所有文档：

```
db.orders.find()
```

现在过滤集合。获取 2010 年 10 月 25 日后的所有订单，即 order_date 大于 2010 年 10 月 25 日。先创建一个日期对象。在 JavaScript shell 中可以这样做：

```
var refdate = new Date(2010, 9, 25);
```

JavaScript 日期中月份从 0 而非 1 开始，所以数字 9 表示 10 月。在 Python 中，要创建同样的变量可以像这样写：

```
from datetime import datetime
refdate = datetime(2010, 10, 25)
```

在 Ruby 里是这样：

```
require 'date'
refdate = Date.new(2010, 10, 25)
```

接着，把 refdate 传入比较器来比较 order_date 字段值和 refdate。查询如下：

```
db.orders.find({"order_date": {$gt: refdate}});
```

MongoDB 支持丰富多样的比较器，包括小于、大于、小于等于、大于等于、等于、不等于。此外，它还支持集合的包含和排除，例如是否包含在或不在给定集合里。

数据集是嵌套文档，所以如果能按嵌套属性值查询会非常有用。在 MongoDB 里这很容易。你可以使用点符号遍历树，访问任何嵌套字段。例如要获取 orders 集合中条目名称为 latter

的所有文档，查询可以这么写：

```
db.orders.find({ "line_items.item.name" : "latte" })
```

无论嵌套的是单值还是列表，点符号（orders 集合的例子）都能用。

MongoDB 的表达式匹配支持使用正则表达式。正则表达式除了可以用于顶层字段外，也可以用于嵌套文档，二者方式相同。

在关系型数据库里，索引是一种非常聪明的用来改进查询速度的方法。它的工作方式非常简单。索引提供基于类 B 树结构的高效查询机制，避免了全表扫描。由于减少了查询相关记录需要查找的数据，因此查询变得更快更高效。

MongoDB 支持用索引提高查询速度。所有集合默认都按_id 值索引。除了默认索引以外，MongoDB 还支持创建二级索引。二级索引可以建在顶层字段或嵌套字段上。例如，可以在条目的数量字段上建一个索引：


```
db.orders.ensureIndex({ "line_items.quantity" : 1 });
```

现在，查询数量为 2 的条目就变得非常快了。试着运行下面的查询：

```
db.orders.find({ "line_items.quantity" : 2 });
```

索引和表分开单独存储，你可能还记得前面章节提到过它们会单独占用命名空间吧？

MongoDB 的数据访问看起来非常简单、丰富和健壮。然而，不是所有 NoSQL 存储都这样，特别是面向列数据库。

5.2.2 用HBase访问数据

HBase 上基于行键的查询是最简单最高效的。HBase 的行键是有序的，且连续的行键存储在一起。因此查找一个行键通常来说就意味着找出起始行键小于等于给定行键的最大有序范围。

这意味着为一个应用程序设计正确的行键极其重要。把行键和数据在语义上关联起来通常是个好主意。在 Google Bigtable 研究论文里，行键是由反向域名组成的，这样一来所有和同一个域名有关的内容就都被组织到了一起。根据这些准则，条目或产品名、订单日期及可能的分类，三者组合作为 orders 表的行键可能就是一个好想法。根据数据最常访问的模式，这三个字段的组合次序会有所变化。如果订单经常按时间顺序访问，那就可以这样创建行键：

```
<date> + <timestamp> + <category> + <product>
```

如果按分类和产品名访问订单最频繁，那就这样创建行键：

```
<category> + <product> + <date> + <timestamp>
```

尽管行键非常重要，并为大量数据的场景提供了高效查询机制，但是二级索引内建的支持却很少。任何没有利用行键的查询都会导致表扫描，而表扫描成本昂贵而且速度缓慢。

第三方工具，例如 Lucene（搜索引擎框架），可以帮助在 HBase 表上建立二级索引。接下来，我们回顾查询数据结构服务器 Redis。

5.2.3　查询Redis

查询 Redis 和向其中插入记录一样简单优雅。前面我们学习了可以通过 get 命令获取特定字符串的值：

```
./redis-cli get akey
```

或者获取列表一个范围里的值：

```
./redis-cli lrange list_of_books 0 4
```

类似地，可以获取一个无序集的成员：

```
./redis-cli smembers asset
```

或者是有序集的成员：

```
./redis-cli zrevrange azset  0 4
```

集合操作（包括求交集、合集和差集）可以分别通过 SINTER、SUNION 和 SDIFF 命令非常容易地执行。

当我们从关系型世界转向 NoSQL 世界时，常听到人们谈论的并不是数据创建或查询，而是数据更新和事务完整性的内容。

接下来，我们将探索在 NoSQL 数据库中数据是如何更新和修改的。

5.3　更新和删除数据

关系型世界深植于 ACID 语义之上，以此为基础提供数据库完整性，并对数据的更新和修改支持不同的隔离级别。与其相反，NoSQL 并不重视 ACID 事务，在某些情况下甚至会完全忽略它。

首先要了解 ACID 的含义。ACID 是 Atomicity、Consistency、Isolation 和 Durability 的首字母缩写，即原子性、一致性、隔离性和持久性。不正式地说，原子性是指交易要么完整执行，要么回滚。一致性是指为使数据库从一种一致状态变为另一种一致状态而对它进行的修改，不存在不一致或混乱的状态。隔离性确保一个操作正在执行时，其他进程不能修改它正在使用的数据。持久性意味着所有已提交的数据都可以从任何系统故障中恢复。

和其他章节一样，我会逐个介绍不同类型的 NoSQL 数据库，首先从 MongoDB 开始。

5.3.1　使用MongoDB、HBase和Redis更新及修改数据

和关系型数据库不同，锁的概念在 NoSQL 存储中不存在。这是一个设计决定，不是巧合。像 MongoDB 这样的数据库的设计初衷就是为了分片和可扩展。这种情况下，锁定多个分布式分片就会变得非常复杂，而且会使数据的更新变得非常缓慢。

尽管缺少锁，却有一些技巧可以帮助实现原子的数据更新。首先，更新整个文档而不是字段。最好使用原子性方法来更新文档。可用的原子性方法包括下面这些。

- $set：设置一个值。
- $inc：按给定量增加一个指定值。
- $push：向数组添加一个值。
- $pushAll：向数组添加一组值。
- $pull：从已有数组中删除一个值。
- $pullAll：从已有数组中删除一组值。

例如，{ $set : { "order_date" : new Date(2010, 10, 01) } }就使用原子性方法更新了 orders 集合中的 order_date 字段。

另一种策略是如果没变就进行更新。这基本上涉及以下三步。

(1) 获取对象。

(2) 在本地修改对象。

(3) 发送这样一个更新请求：如果对象还能匹配旧值的话，就把它更新成这个新值。

文档或行级锁，以及原子性也同样适用于 HBase。

HBase 支持行级别的读写锁。这意味着如果行的任何列值正在被修改、更新或是创建，则行会被锁定。在 HBase 术语里，创建和更新之间的区别并不是很明显。这两种操作都执行类似的逻辑。如果值不存在就是插入操作，否则是更新。

因此行级锁是一个非常棒的注意,但是在空行上获得锁除外,因为它会一直不可用直到超时。

Redis 里支持有限的事务，一个操作可以在这样的事务的范围内执行。Redis 的 MULTI 命令会初始化一个事务。执行 MULTI 以后用 EXEC 执行所有命令，用 DISCARD 回滚操作。下面给出一个简单的例子，我们来看看如何原子地更新两个键：key1 和 key2：

```
> MULTI
OK
> INCR key1
QUEUED
> INCR key2
QUEUED
> EXEC
1) (integer) 1
2) (integer) 1
```

5.3.2 有限原子性和事务完整性

尽管各数据库对原子性的最低支持有所不同，但是它们中许多都有一个相似的特点。本节将介绍围绕 CAP 理论和最终一致性的一些更普遍的思路。

CAP 理论指出以下三个目标中在同一时间只能最大化两个。

- **一致性**（Consistency）：每个客户端都看到同样的数据。
- **可用性**（Availability）：每个客户端都可以读写。
- **分区容忍性**（Partition tolerance）：系统跨越分布式物理网络仍然工作良好。

更多有关 CAP 理论及其对 NoSQL 的影响会在第 9 章中介绍。

另一个经常出现的话题是最终一致性。这个术语有时令人困惑，而且常常没有被正确理解。

最终一致性是用在并行编程和分布式编程领域中的一致性模型。最终一致性可以用如下两种方式来解读。

- 给定足够长的一段时间，期间不发送更新，就可以认为所有更新最终会传播到整个系统，且所有副本都会达到一致。
- 当存在持续的更新时，一个被接受的更新最终要么到达副本，要么重试。

与 ACID 相对，最终一致性意味着基本上可用（Basically Available）、软状态（Soft state）以及最终一致（Eventual consistency）（BASE），这点我们曾在前文提到过。

5.4　小结

本章介绍了 NoSQL 数据库基本的创建、读取、更新和删除操作，探讨了三种 NoSQL 存储的基本操作，即文档存储、面向列存储和键/值哈希映射表。MongoDB 代表文档存储，HBase 代表列存储，Redis 代表键/值哈希映射表。

在讨论中明显发现，对所有数据存储来说，数据的创建或插入比更新更重要。在某些情况下，更新是受限制的。在本章最后，还解释了更新、事务完整性和一致性。

第 6 章

查询 NoSQL 存储

本章内容

- 使用样例数据集演示几种 NoSQL 查询机制
- MongoDB、HBase 和 Redis 的查询用例
- NoSQL 高阶查询
- 支持丰富查询能力的 SQL 替代品

SQL 可能是迄今为止最简单又最强大的领域特定语言了。因为词汇有限、语义明确、语法简单，所以 SQL 简单易学。它非常简洁，使用范围有限，严格按照设计目的运行。它让我们像忍者一样操作结构化数据集。你可以轻松地过滤、排序、切分数据集。基于关系，你可以连接数据集创建交集和合集，也可以汇总数据集，按特定属性进行分组或按分组条件进行过滤。不过 SQL 还是存在一个限制：SQL 基于关系代数，后者只对关系数据库适用。因此，正如其名，NoSQL 中没有 SQL。

SQL 的缺失并不意味着不能查询数据集。毕竟，任何数据的存储目的都是随后可以再获取和操作。对于访问和操作数据，NoSQL 存储有它们自己的方式，前面我们已经见过一些。

NoSQL 实际上是 NonRel，即非关系型。NoSQL 的创建者和支持者之所以远离关系型数据库，是因为关系型数据库强加了结构关系约束和 ACID 事务，特别是它们阻碍了扩展和处理大型数据集，但是他们并不一定反对 SQL。事实上，在 NoSQL 世界里一些人仍然渴望拥有 SQL，结果创建出语法和风格类似经典 SQL 的查询语言。旧习难改，更何况这还是好习惯！

本章中，我们将学到许多查询 NoSQL 存储的技巧。正如前几章通过多个产品和不同技术进行学习一样，本章将从查询 MongoDB 中存储的数据集开始，然后再覆盖 HBase 和 Redis。

6.1　SQL 与 MongoDB 查询功能的相似点

MongoDB 是文档数据库，和关系型数据库几乎没有相似之处，但是 MongoDB 的查询语言感觉非常像 SQL。前面我们已经见过一些简单的例子，所以这儿我假设不需要再举例说明它的类 SQL 查询特性。

要理解 MongoDB 查询语言的能力，看它如何执行，首先得把数据集加载到 MongoDB 数据库中。到现在为止，本书所使用的数据集都比较小，因为重点是介绍 MongoDB 的核心功能，而

不是它在现实场景中的应用。不过在本章中，我要使用一个稍大一些的数据集：MovieLens 数据集，其中包括上百万行电影评分记录。

MovieLens

美国明尼苏达大学计算机科学与工程系的 GroupLens 研究实验室进行的一系列研究包括：

- 推荐系统（Recommender systems）；
- 在线社区（Online communities）；
- 移动和普适技术（Mobile and ubiquitous technologies）；
- 数字图书馆（Digital libraries）；
- 本地地理信息系统（Local geographic information systems）。

MovieLens 数据集是对外可用的 GroupLens 数据集的一部分，它包含了用户对电影的评分。它是一个结构化数据集，提供三个不同的下载包，分别包含 10 万、100 万、1000 万条记录。数据集可以在如下地址下载：grouplens.org/node/73。

首先，去 grouplens.org/node/73 下载包含 100 万条电影评分记录的数据集。可用的下载包包含 `tar.gz`（打包和压缩的）和 `.zip` 压缩格式。选择最合适你的平台的版本。获得压缩包后，把内容解压到一个目录下面。解压完成后，应该得到下面三个文件：

- `movies.dat`
- `ratings.dat`
- `users.dat`

其中 `movies.dat` 数据文件包括电影本身的数据。这个文件包含 3952 条记录，每行一条。记录保存的格式如下：

```
<MovieID>::<Title>::<Genres>
```

`MovieId` 是一个简单的整数序列。Title 是电影标题字符串，里面还包含电影发行年份，列在电影标题后面的圆括号里。电影标题和存储在 IMDB（Internet Movie Database，互联网电影数据库，地址是 www.imdb.com）中的相同。每部电影可能属于多个类型，不同类型用管道符分隔的格式显示。文件的示例行如下：

```
1::Toy Story (1995)::Animation|Children's|Comedy
```

文件 `ratings.dat` 中包含了超过 6000 位用户对那 3952 部电影的评分，评分文件拥有超过 100 万条记录。每行一条，格式如下：

```
UserID::MovieID::Rating::Timestamp
```

`UserID` 和 `MovieID` 分别标识并建立用户和电影的关系。评分 `Rating` 是 5 分制。`Timestamp` 是评分被记录的时间。

`users.dat` 文件是打分用户的数据，其中包含了超过 6000 位用户的信息，格式如下：

```
UserID::Gender::Age::Occupation::Zip-code
```

6.1.1 加载MovieLens数据

为了简单起见，将数据上传到三个 MongoDB 集合中，分别是：movies、ratings 和 users，每个集合各对应一个 .dat 数据文件。mongoimport 工具（www.mongodb.org/display/DOCS/Import+Export+Tools）适合用来把数据从 .dat 文件中提取出来并加载到 MongoDB 文档存储中，但可惜在这儿不能用。因为 MovieLens 数据使用双冒号（::）字符分隔，而 mongoimport 只识别 JSON、逗号分隔、Tab 分隔的格式。

所以我退回到使用编程语言和 MongoDB 驱动来解析文本文件，并加载数据集到 MongoDB 集合中。为了简洁，我选择 Ruby。你还可以使用 Python（也很简洁优雅）、Java、PHP、C 或者其他任何所支持的语言。

如代码清单 6-1 所示的一小段代码，分别从 users、movies 和 ratings 数据文件中提取数据并加载到各自的 MongoDB 集合中。这段代码使用了简单的文件读取和字符串分割功能，以及 MongoDB 驱动来完成任务。代码本身并不是最优雅的。它没有考虑异常处理或者快速处理特大文件的情况，但是能解决眼前的问题。

6

代码清单 6-1 movielens_dataloader.rb

```
require 'rubygems' #Ruby 1.9 里可以跳过这行
require 'mongo'

field_map = {
    "users" => %w(_id gender age occupation zip_code),
    "movies" => %w(_id title genres),
    "ratings" => %w(user_id movie_id rating timestamp)
}

db = Mongo::Connection.new.db("mydb")
collection_map = {
    "users" => db.collection("users"),
    "movies" => db.collection("movies"),
    "ratings" => db.collection("ratings")
}

unless ARGV.length == 1
    puts "Usage: movielens_dataloader data_filename"
    exit(0)
end

class Array
  def to_h(key_definition)
    result_hash = Hash.new()

    counter = 0
    key_definition.each do |definition|
      if not self[counter] == nil then
          if self[counter].is_a? Array or self[counter].is_a? Integer then
              result_hash[definition] = self[counter]
          else
```

```
                result_hash[definition] = self[counter].strip
            end
        else
          # 对每个键插入一个空值，因为可能还需要哈希中包含这些键
          result_hash[definition] = ""
        end
        # 由于某种原因，counter.next 在这里用不了
        counter = counter + 1
      end

      return result_hash
    end
  end

  if File.exists?(ARGV[0])
      file = File.open(ARGV[0], 'r')
      data_set = ARGV[0].chomp.split(".")[0]
      file.each { |line|
          field_names = field_map[data_set]
          field_values = line.split("::").map { |item|
              if item.to_i.to_s == item
                  item = item.to_i
              else
                  item
              end
          }
          puts "field_values: #{field_values}"
          #last_field_value = line.split("::").last
          last_field_value = field_values.last
          puts "last_field_value: #{last_field_value}"
          if last_field_value.split("|").length > 1
             field_values.pop
             field_values.push(last_field_value.split().join('\n').split("|"))
          end
          field_values_doc = field_values.to_h(field_names)
          collection_map[data_set].insert(field_values_doc)
      }
      puts "inserted #{collection_map[data_set].count()} records into the
  #{collection_map[data_set].to_s} collection"
  end
```

movielens_dataloader.rb

数据加载完，就做好查询准备了。查询可以通过 JavaScript shell 或者任何支持的语言来执行。对这个例子来说，大多数查询都用 JavaScript shell 来执行，另外我还选出一小部分例子用其他的编程语言和它们各自的驱动来执行。包含编程语言样例的主要目的是为了说明：大多数（即便不是全部）用 JavaScript shell 实现的查询也可以用其他语言驱动实现。

开始查询前，先启动 MongoDB 服务器，并用 Mongo shell 连接上。这些必要的程序都放在 MongoDB 安装目录的 bin 子目录下。前面章节已经启动过 MongoDB 多次，希望你现在已经非常

熟悉如何启动和停止这些程序了。

在 Mongo JavaScript shell 中，先获取 ratings 集合所有记录的总数，如下所示：

```
db.ratings.count();
```

响应中应该能看到 `1000209`。因为上传了超过 100 万条评分，所以这个数看起来应该是正确的。

接下来，我们用如下的命令获取评分数据的一个样本集：

```
db.ratings.find();
```

在 shell 里面无需显式使用游标来打印集合中的记录。shell 限制了每次返回的行数最大值为 20。要遍历更多数据，只需键入 `it`（`iterate` 的缩写）。`it` 命令的响应包含另外 20 条记录，如果除了在 shell 中已经看到的记录外，还有更多记录，shell 会显示"has more"。

评分数据，比如`{ "_id" : ObjectId("4cdcf1ea5a918708b0000001")`、`"user_id" : 1, "movie_id" : 1193, "rating" : 5, "timestamp" : "978300760" }`，几乎不能带来对相关电影的任何直观的感觉，因为它包含的是电影标识符，不是电影名。要解决这个问题，需要回答下列问题：

6

- 如何获取给定电影的所有评分？
- 如何获取给定评分的电影信息？
- 如何将电影和按电影分组的评分结合起来？

在关系型数据库中，要遍历这些关系会用到联接（join）。在 MongoDB 中，这种关系数据是在服务器范围外显式关联起来的。MongoDB 定义了 DBRef 的概念，用来建立两个不同集合中两个字段之间的关系，但是这种方式存在一些限制，而且和显式的基于标识符的联接也不等同。本节我们不会介绍 DBRef，但是会涵盖前面章节提到的少量 DBRef 的例子，这些例子后面章节也还会再用到。

要获得给定电影的所有评分，可以用电影标识符作为条件过滤数据集。例如，要显示著名的奥斯卡金像奖电影 *Titanic* 的所有评分，需要首先找出它的标识符，然后用这个标识符过滤 ratings 集合。如果不能百分之百地确定电影 *Titanic* 的标题，但是确定标题里包含了单词 titanic，就可以试着近似（而非准确地）匹配 `movies` 集合中的标题字符串。在 RDBMS 中，要在这种情况下找出电影标识符，可能就要在 SQL 的 `where` 子句中使用 `like` 表达式来获取所有候选者的名单。在 MongoDB 中，没有 `like` 表达式，但是有更强大的功能，即用正则表达式来定义匹配模式。要获取 `movies` 集合中所有标题中包括 Titanic 或 titanic 的记录，可以像这样查询：

```
db.movies.find({ title: /titanic/i});
```

这个查询会返回下面的文档集：

```
{ "_id" : 1721, "title" : "Titanic (1997)", "genres" : [ "Drama", "Romance" ] }
{ "_id" : 2157, "title" : "Chambermaid on the Titanic, The (1998)", "genres" :
"Romance" }
{ "_id" : 3403, "title" : "Raise the Titanic (1980)", "genres" : [ "Drama",
"Thriller" ] }
{ "_id" : 3404, "title" : "Titanic (1953)", "genres" : [ "Action", "Drama" ] }
```

MovieLens 数据集的标题字段包含了电影发行年份。在标题字段中，发行年份包含在圆括号中。所以如果记得或者恰好知道电影 *Titanic* 是 1997 年发行的，就可以写出一个更优化的查询表达式：

```
db.movies.find({ title: /titanic.*\(1997\).*/i});
```

这样仅返回一个文档：

```
{ "_id" : 1721, "title" : "Titanic (1997)", "genres" : [ "Drama", "Romance" ] }
```

这个表达式会查找所有包含 Titanic、titanic、TitaniC 或是 TiTAnic 的标题字符串，即忽略大小写。此外，它还会查找字符串（1997）。它还指定在 titanic 和（1997）之间，以及（1997）之后可以有 0 个或多个字符。对正则表达式的支持是非常强大的特性，掌握它永远是值得的。

集合 `ratings` 里 `movie_id` 字段值的范围是由 `movies` 集合的`_id` 字段定义的。所以要获取电影 *Titanic*（标识符为 1721）的所有评分，可以像这样查询：

```
db.ratings.find({ movie_id: 1721 });
```

要找出电影 *Titanic* 的评分总数，可以这样做：

```
db.ratings.find({ movie_id: 1721 }).count();
```

返回结果是 1546。评分是 5 分制的。要获取电影 *Titanic* 所有 5 星评分的列表和总数，可以进一步过滤记录集如下：

```
db.ratings.find({ movie_id: 1721, rating: 5 });

db.ratings.find({ movie_id: 1721, rating: 5 }).count();
```

查询文档的数据类型敏感性

MongoDB 查询文档是数据类型敏感的。也就是说`{ movie_id: "1721"}`和`{ movie_id: 1721 }`并不一样，第一个会匹配字符串而第二个会把值看作数字。当声明文档时，一定要使用正确的数据类型。为了进一步说明，举个例子，在 `ratings` 和 `movies` 集合中 `movie_id` 是作为数字（整数）存储的，因此按字符串查询不会返回正确结果。所以对查询 `db.ratings.find({movie_id: 1721 })`，响应返回总共 1546 个文档，而对 `db.ratings.find({ movie_id: "1721"})`，则没有返回任何文档。

仔细浏览代码清单 6-1，会注意到下面的代码：

```
field_values = line.split("::").map { |item|
            if item.to_i.to_s == item
                item = item.to_i
            else
                item
            end
        }
```

这段代码检查被分割的字符串中是否包含整数值，如果包含就按整数存储。付出这一点额外的努力，将数值型字符串保存为数字有它的好处。在数值型记录上做索引和查询通常比在字符（字符串）记录上做更快更高效。

接下来，你可能希望获取一些 *Titanic* 评分的统计信息。要找出用户给出的所有不同的评分（1 到 5 的可能的整数集合），可以查询如下：

```
db.runCommand({ distinct: 'ratings', key: 'rating', query: { movie_id: 1721} });
```

电影 *Titanic* 的评分包括了从 1 到 5 所有可能的情况，所以结果是这样的：

```
{ "values" : [ 1, 2, 3, 4, 5 ], "ok" : 1 }
```

runCommand 接受下面的参数：

- distinct 字段指定集合名。
- key 字段指定需要列出唯一值的字段。
- query 字段可选，用于过滤集合。

runCommand 和你到目前为止所见过的其他查询模式有一点点不同，这是因为在查找唯一值前，先对集合进行了过滤。集合中所有评分的唯一值可以用我们已知的方式列出来，如下所示：

6

```
db.ratings.distinct("rating");
```

上面我们已知对 *Titanic* 的评分包括从 1 到 5 所有可能的值。要想进一步了解其中每个评分，可以对每个分值分别汇总如下：

```
db.ratings.group(
... { key: { rating:true },
...   initial: { count:0 },
...   cond: { movie_id:1721 },
...   reduce: function(obj, prev) { prev.count++; }
... }
... );
```

分组查询的输出是如下的数组：

```
[
    {
        "rating" : 4,
        "count" : 500
    },
    {
        "rating" : 1,
        "count" : 100
    },
    {
        "rating" : 5,
        "count" : 389
    },
    {
        "rating" : 3,
        "count" : 381
    },
    {
        "rating" : 2,
        "count" : 176
    }
]
```

分组函数在单个 MongoDB 实例上用起来非常方便，但在分片部署环境中无法使用。在分片的 MongoDB 部署环境中，要使用 MongoDB 的 MapReduce 工具来运行分组函数。等我解释完分组操作，会给出一个 MapReduce 版本的分组函数。

分组操作的输入是一个对象，它包括以下字段。

- Key：分组用的字段。前面例子里只用了一个字段 `rating`。额外的字段可以用逗号分隔的列表包含进来，共同作为 key 字段的值，例如 `key: { fieldA: true, fieldB: true}`。
- initial：聚合的初始值。前面例子里初始的计数值（count）为 0。
- cond：过滤集合的查询文档。
- reduce：聚合函数。
- keyf（可选）：当所需的键不包含在已有文档中时，用来替代它的派生键。
- finalize（可选）：在聚合函数遍历的每个条目上运行的函数。可用来修改已有条目。

理论上，前面的例子应该能很容易改成对每部电影按分值进行分组，只要使用下面的分组操作即可：

```
db.ratings.group(
... { key: { movie_id:true, rating:true },
...   initial: { count:0 },
...   reduce: function(obj, prev) { prev.count++; }
... }
... );
```

但是现实是，这个操作对拥有 100 万记录的 `ratings` 集合不可行，你会得到下面这样的错误信息：

```
Fri Nov 12 14:27:03 uncaught exception: group command failed: {
    "errmsg" : "exception: group() can't handle more than 10000 unique keys",
    "code" : 10043,
    "ok" : 0
}
```

结果要以单个 BSON 对象的形式返回，因此分组操作执行的集合中键的个数不能超过 1 万。这个限制可以通过 MapReduce 工具来克服。

下一节将介绍 MongoDB 的 MapReduce 工具，还会在整个评分数据集上执行一些聚合函数。

6.1.2　MongoDB中的MapReduce

MapReduce 是 Google 申请了专利的软件框架，支持在大型分布式集群上进行分布式计算。你可以在题为“MapReduce: Simplified Data Processing on Large Clusters”的研究论文中阅读有关 Google MapReduce 的内容，地址是：http://labs.google.com/papers/mapreduce.html。

Google MapReduce 框架为开源社区中许多相似产品和分布式计算框架带来了灵感，MongoDB 就是其中一个。Google 和 MongoDB 的 MapReduce 功能同样从函数式编程世界的类似结构中得到了启发。在函数式编程中，map 函数应用于集合的每个成员，而 reduce 函数或者 fold 函数在整个集合上进行聚合。

> MongoDB 的 MapReduce 功能与 Google 的 MapReduce 并不完全相同。Hadoop 的 MapReduce 是 Google 分布式计算理念的开源实现，而且同时支持列数据库（HBase）和基于 MapReduce 的计算。

掌握 MapReduce 可能听起来是件挺难的事儿，不过一旦明白了它的结构和流程，MapReduce 就能变成非常好的助手，可以帮助我们在分布式的数据集合上执行大规模计算。所以我们最好先从简单的例子开始，然后接触更复杂的例子，这样学习起来比较容易，并且更容易掌握主题。

最简单的聚合的例子恐怕就是对集合中的条目进行计数了。要使用 MapReduce，首先要定义 map 函数和 reduce 函数，然后在集合上执行 map 和 reduce 函数。map 函数对集合中的每个元素都执行同一个函数，并返回一个键/值对作为输出。然后 reduce 函数用 map 函数输出的键/值对做为输入，在所有键/值对上执行聚合函数，得到最终结果。

6

计算 users 集合中女性受访者（F）和男性受访者（M）数量的 map 函数如下：

```
> var map = function() {
... emit({ gender:this.gender }, { count:1 });
... };
```

movielens_queries.txt

这个 map 函数对集合中的每个条目生成一个键/值对，其中包含 gender 属性和计数，计数定值为 1。

reduce 函数用来计算所有用户中出现的男性和女性总数，定义如下：

```
> var reduce = function(key, values) {
... var count = 0;
... values.forEach(function(v) {
... count += v['count'];
... });
...
... return { count:count };
... };
```

movielens_queries.txt

reduce 函数接受 map 函数输出的键/值对。在这个特定的 reduce 函数中，每个键/值对的值被传递给一个用于计算（特定性别）总数的函数。借助 JavaScript 可以把对象成员和值当作哈希数据结构来访问的能力，代码 count += v['count']也可以写作 count += v.count。

最后，对 users 集合运行 map 和 reduce 函数，产生 users 集合中女性和男性总数的输出。mapReduce 运行和 result 解析命令如下：

```
> var ratings_respondents_by_gender = db.users.mapReduce(map, reduce);
> ratings_respondents_by_gender
{
    "result" : "tmp.mr.mapreduce_1290399924_2",
    "timeMillis" : 538,
    "counts" : {
```

```
        "input" : 6040,
        "emit" : 6040,
        "output" : 2
    },
    "ok" : 1,
}
> db[ratings_respondents_by_gender.result].find();
{ "_id" : { "gender" : "F" }, "value" : { "count" : 1709 } }
{ "_id" : { "gender" : "M" }, "value" : { "count" : 4331 } }
```

movielens_queries.txt

为了验证输出，分别按性别值 F 和 M 过滤 users 集合，然后对每个过滤出来的子集的文档进行计数。按性别 F 和 M 对 users 集合进行过滤并计数的命令如下：

```
> db.users.find({ "gender":"F" }).count();
1709
> db.users.find({ "gender":"M" }).count();
4331
```

movielens_queries.txt

接下来稍微修改一下 map 函数，然后在 ratings 集合上运行 map 和 reduce 函数，获取每部电影每个评分（1、2、3、4 和 5）的总数，即按每部电影的评分分组计数。下面是在 ratings 集合上运行的 map 和 reduce 函数的完整定义：

```
> var map = function() {
... emit({ movie_id:this.movie_id, rating:this.rating }, { count:1 });
... };
> var reduce = function(key, values) {
... var count = 0;
... values.forEach(function(v) {
... count += v['count'];
... });
...
... return { count: count };
... };
> var group_by_movies_by_rating = db.ratings.mapReduce(map, reduce);
> db[group_by_movies_by_rating.result].find();
```

movielens_queries.txt

要获取电影 *Titanic*（标识符 movie_id 为 1721）每种评分的总数，只需用嵌套属性访问方法过滤 MapReduce 的输出，像这样：

```
> db[group_by_movies_by_rating.result].find({ "_id.movie_id":1721 });
{ "_id" : { "movie_id" : 1721, "rating" : 1 }, "value" : { "count" : 100 } }
{ "_id" : { "movie_id" : 1721, "rating" : 2 }, "value" : { "count" : 176 } }
{ "_id" : { "movie_id" : 1721, "rating" : 3 }, "value" : { "count" : 381 } }
{ "_id" : { "movie_id" : 1721, "rating" : 4 }, "value" : { "count" : 500 } }
{ "_id" : { "movie_id" : 1721, "rating" : 5 }, "value" : { "count" : 389 } }
```

movielens_queries.txt

到目前为止的两个 MapReduce 例子中，reduce 函数相同，map 函数不同。所有例子里不同的键/值对都生成计数值 1。一个例子中为每个文档生成的键/值对包含性别属性，另一个例子为文档生成的键/值对的键是电影标识符和评分标识符的组合。

下面，计算 ratings 集合中每部电影的平均评分：

```
> var map = function() {
... emit({ movie_id:this.movie_id }, { rating:this.rating, count:1 });
... };

> var reduce = function(key, values) {
... var sum = 0;
... var count = 0;
... values.forEach(function(v) {
... sum += v['rating'];
... count += v['count'];
... });
...
... return { average:(sum/count) };
... };
> var average_rating_per_movie = db.ratings.mapReduce(map, reduce);
> db[average_rating_per_movie.result].find();
```

movielens_queries.txt

MapReduce 支持编写多种复杂的聚合算法，本节只展示了一小部分，还有一些将在本书后面的部分里介绍。

现在你已经了解了多种 MongoDB 集合的查询方式。下面我们熟悉一下表格型数据库的查询，我们将用 HBase 来演示查询机制。

6.2　访问 HBase 等面向列数据库中的数据

在开始查询 HBase 前，需要先存一些数据进去。和 MongoDB 一样，前面我们简单了解过使用 HBase 存储和访问数据的方法，包括它的底层文件系统，这个通常默认是 Hadoop 分布式文件系统（HDFS）。我们还了解了 HBase 和 Hadoop 的基础知识。本节是以这些知识为基础的。为了让例子真实可用，我们将 1970 年至 2010 年 2 月 NYSE 每日股票市场的历史数据加载到 HBase 实例中，然后用 HBase 的查询机制访问这些数据。这些历史数据是从 Infochimp.org 的原始数据里整理出来的，可以通过下面的地址访问：www.infochimps.com/datasets/nyse-daily-1970-2010-open-close-high-low-and-volume。

历史市场数据

整个数据集的 zip 包大概有 199MB，不过按 HDFS 和 HBase 的标准离“大”还很远。HBase 和 Hadoop 基础设施能够而且经常被用来处理 PB 级、跨越多台物理机的数据。这个例子里我特意选择了一个易于管理的数据集，主要是为了避免因准备和加载大数据集的事情而分心。本章的主要内容是 NoSQL 存储的查询方法，这一节重点关注面向列数据库。使用更小的数据集来学习

数据访问方法更可控，而且学到的概念也同样适用于更大量的数据。

样例的数据字段逻辑上分为以下三个不同种类。

- 外汇、股票代码和日期的组合作为唯一标识。
- 开盘、高、低、收盘价及调整过的收盘价作为价格的考量。
- 每日成交量。

行键可以用外汇、股票代码和日期的组合来创建。这样 NYSE、AA、2008-02-27 可以构成行键 NYSEAA20080227。所有价格相关的信息可以存储在名为 `price` 的列族中，而交易量可以存储在 `volume` 列族中。

表名设为 `historical_daily_stock_price`。要获得 NYSE、AA、2008-02-27 的记录数据，可以这样查询：

```
get 'historical_daily_stock_price', 'NYSEAA20080227'
```

可以这样获取开盘价格：

```
get 'historical_daily_stock_price', 'NYSEAA20080227', 'price:open'
```

也可以用编程语言来查询数据。下面是一个 Java 样例程序，用来获取开盘和高点的价格数据：

```
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.io.RowResult;

import java.util.HashMap;
import java.util.Map;
import java.io.IOException;

public class HBaseConnector {

public static Map retrievePriceData(String rowKey) throws IOException {
HTable table = new HTable(new HBaseConfiguration(),
 "historical_daily_stock_price");
Map stockData = new HashMap();

RowResult result = table.getRow(rowKey);

for (byte[] column : result.keySet()) {
    stockData.put(new String(column), new
String(result.get(column).getValue()));
}

return stockData;
}

public static void main(String[] args) throws IOException {
    Map stock_data = HBaseConnector.retrievePriceData("NYSEAA20080227");
    System.out.println(stock_data.get("price:open"));
    System.out.println(stock_data.get("price:high"));
}

}
```

HBaseConnector.java

除了我们已经演示过的部分以外，基本上 HBase 再没包含多少高级的查询技术了，但是它的索引和查询能力可以借助 Lucene 和 Hive 进行扩展。使用 Hive 和 HBase 的细节会在第 12 章里演示。

6.3 查询 Redis 数据存储

和 MongoDB、HBase 一样，前面章节里我们也用过了 Redis。在过去的几章里我们学习了在 Redis 中存储和访问数据的基本知识。本节将更进一步深入数据查询主题。和到目前为止的其他演示一样，首先要加载样例数据集到 Redis 实例中。

为了演示，我用到了可以从 www.nyc.gov/data 在线获取的停车设施 NYC 数据挖掘公共源数据。可下载的数据是 CSV 格式，名为 `parking_facilities.csv`。代码清单 6-2 的 Python 程序可以用来解析 CSV 格式数据，并将其加载入本地 Redis 存储。记得在运行 Python 脚本来加载数据前，先启动 Redis 安装目录里的 Redis-server 程序，它会启动 Redis 服务器实例，默认监听端口 6379 上的客户端连接。

代码清单 6-2 解析 NYC 停车设施数据的 Python 程序

```
import csv
import redis

f = open("parking_facilities.csv", "r")
parking_facilities = csv.DictReader(f, delimiter=',')
r = redis.Redis(host='localhost', port=6379, db=0)

def add_parking_facility(license_number,
        facility_type,
        entity_name,
        camis_trade_name,
        address_bldg,
        address_street_name,
        address_location,
        address_city,
        address_state,
        address_zip_code,
        telephone_number,
        number_of_spaces):
    if r.sadd("parking_facilities_set", license_number):
        r.hset("parking_facility:%s" % license_number, "facility_type",
facility_type)
        r.hset("parking_facility:%s" % license_number, "entity_name",
entity_name)
        r.hset("parking_facility:%s" % license_number, "camis_trade_name",
camis_trade_name)
        r.hset("parking_facility:%s" % license_number, "address_bldg",
address_bldg)
        r.hset("parking_facility:%s" % license_number, "address_street_name",
address_street_name)
        r.hset("parking_facility:%s" % license_number, "address_location",
```

```
address_location)
        r.hset("parking_facility:%s" % license_number, "address_city",
address_city)
        r.hset("parking_facility:%s" % license_number, "address_state",
address_state)
        r.hset("parking_facility:%s" % license_number, "address_zip_code",
address_zip_code)
        r.hset("parking_facility:%s" % license_number, "telephone_number",
telephone_number)
        r.hset("parking_facility:%s" % license_number, "number_of_spaces",
number_of_spaces)
        return True
    else:
        return False


if __name__ == "__main__":
    for parking_facility_hash in parking_facilities:
        add_parking_facility(parking_facility_hash['License Number'],
            parking_facility_hash['Facility Type'],
            parking_facility_hash['Entity Name'],
            parking_facility_hash['Camis Trade Name'],
            parking_facility_hash['Address Bldg'],
            parking_facility_hash['Address Street Name'],
            parking_facility_hash['Address Location'],
            parking_facility_hash['Address City'],
            parking_facility_hash['Address State'],
            parking_facility_hash['Address Zip Code'],
            parking_facility_hash['Telephone Number'],
            parking_facility_hash['Number of Spaces'])
        print "added parking_facility with %s" % parking_facility_hash['License
Number']
```

nyc_parking_data_loader.py

这个 Python 程序遍历提取出的哈希记录列表，然后把值保存到 Redis 实例中。每个哈希记录以许可证号为键。所有许可证号都保存在名为 parking_facilities_set 的集合中。

要获取 parking_facilities_set 里所有的许可证号，用另一个程序或是命令行客户端连上服务器，然后执行下面的命令：

```
SMEMBERS parking_facilities_set
```

集合中全部 1912 条许可证号都会打印出来。可以运行 wc -l paking_facilities.csv 确认数字是否正确。CSV 中的每行都对应一个停车设施，所以两个数字应该一致。

每个停车设施的属性都保存在一个哈希中，由形如 parking_facility:<license_number>的键来标识。所以，要找出许可证号 1105006 关联的哈希的所有键，可以使用下面这个命令：

```
HKEYS parking_facility:1105006
```

响应如下：

```
1. "facility_type"
2. "entity_name"
3. "camis_trade_name"
4. "address_bldg"
5. "address_street_name"
6. "address_location"
7. "address_city"
8. "address_state"
9. "address_zip_code"
10. "telephone_number"
11. "number_of_spaces"
```

许可证号 1105006 是命令 SMEMBERS parking_facilities_set 返回的列表中的第一个。不过无序集没有顺序，所以重新执行这个命令，同一个许可证号可能不会再排第一。如果需要列表按一定的顺序显示，应该用有序集代替无序集，用下面这行代码换掉 if r.sadd("parking_facilities_set", license_number):这行就可以：


```
if r.zadd("parking_facilities_set", license_number):
```

现在，可以查询哈希中的特定值，比如说设施类型：

```
HGET parking_facility:1105006 facility_type
```

结果是"Parking Lot"。可以用 HVALS 命令打印出所有值，如下：

```
HVALS parking_facility:1105006
```

结果是：

```
1. "Parking Lot"
2. "CENTRAL PARKING SYSTEM OF NEW YORK, INC"
3. ""
4. "41-61"
5. "KISSENA BOULEVARD"
6. ""
7. "QUEENS"
8. "NY"
9. "11355"
10. "2126296602"
11. "808"
```

当然，如果能显示出一个哈希中所有键和对应的值就更好了。可以使用如下的 HGETALL 命令实现：

```
HGETALL parking_facility:1105006
```

响应如下：

```
1. "facility_type"
2. "Parking Lot"
3. "entity_name"
4. "CENTRAL PARKING SYSTEM OF NEW YORK, INC"
5. "camis_trade_name"
6. ""
7. "address_bldg"
8. "41-61"
9. "address_street_name"
```

```
10. "KISSENA BOULEVARD"
11. "address_location"
12. ""
13. "address_city"
14. "QUEENS"
15. "address_state"
16. "NY"
17. "address_zip_code"
18. "11355"
19. "telephone_number"
20. "2126296602"
21. "number_of_spaces"
22. "808"
```

有时，并不需要所有键/值对，而只需要打印出特定一组字段的值。例如可能只需要打印出 address_city 和 address_zip_code 字段的值：

```
HMGET parking_facility:1105006 address_city address_zip_code
```

响应是：

```
1. "QUEENS"
2. "11355"
```

类似地，也可以用命令 HMSET 设置一组字段的值。要获取键的总数，可以用 HLEN 命令：

```
HLEN parking_facility:1105006
```

响应是 11。如果想要检查 address_city 是不是这些键中的一个，可以使用 HEXISTS 命令检查它是否存在：

```
HEXISTS parking_facility:1105006 address_city
```

响应 1 表示字段存在，0 表示不存在。

再回到集合 parking_facilities_set 上，有可能你需要的只是成员总数，而不是列出所有成员，此时可以使用 SCARD 命令：

```
SCARD parking_facilities_set
```

结果是 1912。要检查一个成员是否存在于无序集中，可以使用命令 SISMEMBER。例如要检查 1005006 是否是无序集的成员，可以使用下面的命令：

```
SISMEMBER parking_facilities_set 1105006
```

整数值 0 和 1 分别代表一个成员不属于无序集（false）、属于无序集（true）。

6.4　小结

本章举例说明了一些更高级的查询机制。MongoDB 查询是通过电影评分数据集示例来解释的，HBase 查询是用历史股市数据样例来阐释的，而 Redis 查询能力则是用 NYC 政府样例数据说明的。

对查询功能的覆盖谈不上详尽，并没有覆盖所有类型的用例。本章演示的用例只是无数可能方式中有限的一部分。然而通过浏览这些例子，应该能帮助你熟悉 NoSQL 数据查询的风格和机制。

第 7 章

修改数据存储及管理演进

本章内容

- 使用文档数据库、面向列数据库和键/值存储来管理数据的 schema
- 数据演进与存储维护
- 数据导入与导出

随着时间的推移，数据会演进并且发生变化，有时变化会非常迅猛，有时节奏缓慢，而且不那么激进。另外，数据往往比单个程序存活得更久。而且就算是根据特定用例来设计，数据常常也会以从未设想过的方式被使用。

即便如此，关系数据库通常不太关注数据的演变。虽然它确实提供了修改 schema 和数据类型的方法，但是仍然假定在大多数情况下，元数据会保持静态不变。它假定对大多数类型的数据来说，结构普遍都是统一的，它认为首先要确保数据 schema 的正确性。关系型数据库关注如何高效地存储结构化的、致密的数据集，因此记录的正规化（normalization）就非常重要。

虽然本章讨论的不是 RDBMS 能否适应改变，不过应当注意到，修改数据 schema 和数据类型、合并两个不同版本 schema 的数据，这些事情在 RDBMS 中通常都比较复杂，往往还要用到很多权宜之计。比如像添加新列到已有表（即已经存在一些数据），这么温和的改变也可能会造成严重的问题，特别是当新列还需要保存唯一值时。对这类问题，应对之道是有的，但它们并不优雅，或者不是无缝的解决方案。与之相反，许多 NoSQL 数据库推崇无 schema 数据存储，这样变化起来就更自然、更容易。

和前面章节一样，我们通过三类流行的 NoSQL 产品来探讨数据库修改与演变的话题，它们分别是：

- 文档数据库
- 列数据库
- 键/值存储

7.1　修改文档数据库

文档数据库形式上是无 schema 的，它支持在集合中存储自包含的文档作为记录或是条目。文档数据库并非那么坚持正规的 schema 或形式，所以它明显更容易接纳变化和修改。事实上，

文档数据库并不会阻止你把完全不同的文档存储到同一个集合里，虽然这样的集合逻辑上可能不是很有用。

CouchDB（现在作为 Couchbase 的一部分）和 MongoDB 是领先的开源文档数据库，在存储不同属性的文档到同一个集合里这一点上，都表现得极为灵活。例如，你可以很容易地把下面两个文档存储在一起：

```
{ name => "John Doe", organization => "Great Co", email => "john.doe@example.com" }
{ name => "Wei Chin", company => "Work Well", phone => "123-456-7890" }
```

启动 CouchDB，然后把这两个文档存储在名为 contacts 的数据库里。

> 一个 MongoDB 服务器可以托管多个数据库，每个数据库又可以有多个集合。相对地，一个 CouchDB 服务器可以托管多个服务器，但没有集合的概念。

可以使用命令行工具 Futon 或者其他外部程序来和 CouchDB 交互，并存储文档。

> CouchDB 提供了 REST API 来执行所有工作，包括创建、管理数据库和文档，甚至是触发复制。在继续下面内容之前，请先安装 CouchDB。实践出真知，没有什么比自己尝试各种例子、试验各种概念更好的了。如果安装设置时需要帮助，请参阅附录 A。附录 A 中包含了本书中所有 NoSQL 产品的安装和设置指南。

图 7-1 是用 Futon 查看 contracts 数据库中两个文档列表的截图。列表中只显示了 id（CouchDB 生成的 UUID）和文档版本号。



图　7-1

展开 Wei Chin 的信息会显示出包括所有字段的完整文档，如图 7-2 所示。



图　7-2

图 7-2 底部右侧的导航键可以让你浏览文档的不同版本。这在很多数据库中可能都看不到，而且会让人想起版本控制软件或文档管理系统，不过它是内置在 CouchDB 里的一个非常重要的特性。在 CouchDB 里，对文档的更新在底层会被翻译成增加一个文档的新版本。因此，如果把名字从“Wei Chin”改成“Wei Lee Chin”，则更新后，当前版本文档（JSON 格式）会像下面这样：

```
{
   "_id": "797f603b2d043f6b23264e27fa00121f",
   "_rev": "2-949a21d63459638cbd392e6b3a27989d",
   "name": "Wei Lee Chin",
   "company": "Work Well",
   "phone": "123-456-7890"
}
```

couchdb_example.txt

除了 name 字段值的更新，还能看到`_rev` 属性值的变化。`_rev` 字段存储着文档版本号。原来的版本号是 `1-63726a5e55e33ed02a927ca8518df073`，更新以后的版本号是 `2-949a21d63459638cbd392e6b3a27989d`。CouchDB 中的版本号格式形如 `N-<hash value>`，其中 N 表示文档被更新的次数，哈希值是文档传输格式的 MD5 哈希值。文档刚创建时，N 是 1。

7

> MD5 是单向散列算法，接受任意长度的数据，生成 128 位指纹或信息摘要。更多有关 MD5 的信息请访问：www.ietf.org/rfc/rfc1321.txt。

CouchDB 使用 MVCC（MultiVersion Concurrency Control，多版本并发控制）以方便并发访问数据库。采用 MVCC 使得 CouchDB 能够避免引入锁机制（以确保写操作准确）。每个文档都有自己的版本，文档版本能帮助解决冲突。在更新一个文档前，首先要确认当前版本（更新以前的版本）和被读取时的版本一样。如果版本不匹配，表明可能存在冲突，比如在读取和随后的更新之间，另一个独立线程进行了更新。在更新文档时，会保存整个新文档，而不是更新已存在的文档。这个过程附带的好处是性能会有所提升，因为向连续内存中追加的速度比原地更新更快。由于版本（或者说版本号）是 CouchDB 中的核心概念，因此你会看到多个版本。

不过默认情况下，文档的各个版本并不永久保存。版本控制的目的是为了避免冲突，同时提供并发能力。压缩和复制会删除旧版本，任何时刻只有最新版本的文档才确定存在。这意味着默认情况下无法使用`_rev` 字段查询或访问旧版本的文档。在单节点的场景中，你可能会试图通过关闭压缩来保存旧版本。但是这个策略放到集群上就会立即失效，因为只有最新版本才会被复制。

如果确实需要保存多个版本并查询旧版本的文档，就要通过编程来实现。CouchDB 创始人对这个问题有一个简单高效的解决方案，可以参阅 http://blog.couchone.com/post/632718824/simple-document-versioningwith-couchdb。这个办法非常直接，它建议按以下步骤来做。

- 在访问当前版本的文档时，提取出一个文档的字符串表示。
- 更新前，用 Base64 编码字符串，将其二进制形式作为文档的附件保存下来。使用当前版

本号（更新前的版本）作为附件名。

这意味着，如果一个文档是这样访问的：

http://127.0.0.1:5984/contacts/797f603b2d043f6b23264e27fa00121f

则包含此文档作为附件的版本 2 可以这样访问：

http://127.0.0.1:5984/contacts/797f603b2d043f6b23264e27fa00121f/2-949a21d63459638cbd392e6b3a27989d

大多数情况下，这种管理版本的方式既简单又好用。更复杂的版本管理系统可以根据使用情况来保存文档的版本。

Futon 就是用前面演示过的技术来管理文档版本的。这个技术在 CouchDB 的 jQuery JavaScript 客户端类库里得到了实现。jQuery 客户端类库可以在这里获取：http://svn.apache.org/viewvc?revision=948262&view=revision。

所以，尽管 CouchDB 中的版本是一个非常有趣的特性，但文档存储的可扩展性和灵活性却是更普遍的特性，这也能在其他弱 schema 的 NoSQL 数据库中找到。

7.1.1　弱schema的灵活性

从前面的例子里明显看出，CouchDB 完全能够在同一个数据库中存储两个字段不同的文档。这样做有很多好处，特别是在以下情况中：

- 高效存储稀疏数据集，因为值为 null 的字段不占用存储。
- 随着文档结构发生变化，增加额外字段的工作显得微不足道。

在前面例子里，`"John Doe"`有一个 email 地址，但是`"Wei Chin"`没有。这不是问题，他们可以共存于同一个数据库中。将来要是`"Wei Chin"`也有了 email 地址，比如说`wei.chin@example.com`，那么无需任何多余开销就能将新字段添加到文档里。同样地，字段也可以被删除，字段值也可以被修改。

除了可以添加和删除字段，对字段的数据类型也没有什么强制规则。所以一个存储字符串值的字段也可以存储整数，甚至还可以存储数组类型。这意味着不用担心强类型。另一方面，这意味着应用程序需要确保数据是验证过的，而且值的语义是一致的。

到目前为止，已经通过 CouchDB 初步解释了弱 schema 的灵活性。为了展示这种灵活性的其他方面，下面使用 MongoDB。首先创建一个名为 `contacts` 的 MongoDB 集合，并添加两个文档到集合中。启动 MongoDB 服务器，然后按顺序执行下面的命令：

```
use mydb
db.contacts.insert({ name:"John Doe", organization:"Great Co",
 email:"john.doe@example.com" });
db.contacts.insert({ name:"Wei Chin", company:"Work Well", phone:"123-456-7890"
 });
```

mongodb_example.txt

接下来，确认集合已经创建好，而且两个文档也在里面。可以像下面这样仅仅列出文档来确认：

```
db.contacts.find();
```

mongodb_example.txt

查询结果应该像这样：

```
{ "_id" : ObjectId("4d2bbad6febd3e2b32bed964"), "name" : "John Doe",
"organization" : "Great Co", "email" : "john.doe@example.com" }
{ "_id" : ObjectId("4d2bbb43febd3e2b32bed965"), "name" : "Wei Chin", "company" :
"Work Well", "phone" : "123-456-7890" }
```

`_id` 值可能会不一样，因为这些是 MongoDB 在我的系统上生成的，在不同实例上当然会有所变化。现在，添加一个新字段 `email` 到`"Wei Chin"`的文档中：

```
var doc = db.contacts.findOne({ _id:ObjectId("4d2bbb43febd3e2b32bed965") });
doc.email = "wei.chin@example.com";
db.contacts.save(doc);
```

mongodb_example.txt

用`_id` 来获取文档，然后赋值给 `email` 字段并保存文档。要确认已经添加了新字段，只要从 `contacts` 集合中重新获取文档即可：


```
db.contacts.find();
```

mongodb_example.txt

结果如下：

```
{ "_id" : ObjectId("4d2bbad6febd3e2b32bed964"), "name" : "John Doe",
"organization" : "Great Co", "email" : "john.doe@example.com" }
{ "_id" : ObjectId("4d2bbb43febd3e2b32bed965"), "name" : "Wei Chin", "company":
"Work Well", "phone" : "123-456-7890", "email" : "wei.chin@example.com" }
```

与 CouchDB 不同，MongoDB 不会维护文档版本，而会更新文档。

现在，比如说有另一个集合 `contacts2`，里面包含一些文档，我们需要将两个集合 `contacts` 和 `contacts2` 合二为一。你会怎么做？

很可惜目前还没有一个神奇按钮或是命令可以实现一键合并多个集合，不过用你喜欢的语言写一个小脚本来合并两个集合并不是非常困难。在设计合并脚本时，有如下几点值得考虑。

- 用一个开关来决定当需要合并的集合中存在`_id` 相等的文档时，应覆写、更新还是复制。同一个集合中任意两个文档不能有相同的`_id` 值。覆写意味着第二个集合的文档会覆盖第一个集合中相应的文档。更新和复制又是其他的合并策略。
- 根据非`_id` 字段进行合并。

项目 mongo-tools 中包括了用来合并两个 MongoDB 集合的 ruby 脚本，可以在这里访问到：https://github.com/tshanky/mongo-tools。

7.1.2 MongoDB的数据导入与导出

数据导出和导入在数据库备份、恢复和合并中是很重要的一步，而且很常用。在这方面，

MongoDB 提供了一些有用的工具来协助用户。

1. mongoimport

如果需要导入的数据就放在一个文件里，而且是 JSON 格式、CSV 格式或者 TSV 格式等文本格式，可以用 mongoimport 将数据导入到 MongoDB 集合中。它有一些选项值得了解一下，不带参数执行命令就会列出这些选项。不带参数运行 bin/monogimport，输出如下：

```
connected to: 127.0.0.1
no collection specified!
options:
  --help                  produce help message
  -v [ --verbose ]        be more verbose (include multiple times for more
                          verbosity e.g. -vvvvv)
  -h [ --host ] arg       mongo host to connect to ("left,right" for pairs)
  --port arg              server port. Can also use --host hostname:port
  -d [ --db ] arg         database to use
  -c [ --collection ] arg collection to use (some commands)
  -u [ --username ] arg   username
  -p [ --password ] arg   password
  --ipv6                  enable IPv6 support (disabled by default)
  --dbpath arg            directly access mongod database files in the given
                          path, instead of connecting to a mongod  server -
                          needs to lock the data directory, so cannot be used
                          if a mongod is currently accessing the same path
  --directoryperdb        if dbpath specified, each db is in a separate
                          directory
  -f [ --fields ] arg     comma separated list of field names e.g. -f name,age
  --fieldFile arg         file with fields names - 1 per line
  --ignoreBlanks          if given, empty fields in csv and tsv will be ignored
  --type arg              type of file to import.  default: json (json,csv,tsv)
  --file arg              file to import from; if not specified stdin is used
  --drop                  drop collection first
  --headerline            CSV,TSV only - use first line as headers
  --upsert                insert or update objects that already exist
  --upsertFields arg      comma-separated fields for the query part of the
                          upsert. You should make sure this is indexed
  --stopOnError           stop importing at first error rather than continuing
  --jsonArray             load a json array, not one item per line. Currently
                          limited to 4MB.
```

这个导入工具很有用，但是如果要导入比 CSV 或 TSV（或 JSON 格式）稍微复杂一点的数据，它就达到极限了。还记得第 6 章里用来加载 MovieLens 数据成 MongoDB 集合的 Ruby 脚本吗？mongoimport 可没法完成那个任务。

2. mongoexport

和加载数据正好相反的是导出数据。如果 JSON 或 CSV 格式能满足你的需要，就可以使用这个工具把数据从集合里导出来。要了解有哪些可用的选项，不带任何参数运行 mongoexport 即可。所有选项如下：

```
connected to: 127.0.0.1
no collection specified!
options:
  --help                  produce help message
```

```
  -v [ --verbose ]         be more verbose (include multiple times for more
                           verbosity e.g. -vvvvv)
  -h [ --host ] arg        mongo host to connect to ("left,right" for pairs)
  --port arg               server port. Can also use --host hostname:port
  -d [ --db ] arg          database to use
  -c [ --collection ] arg  collection to use (some commands)
  -u [ --username ] arg    username
  -p [ --password ] arg    password
  --ipv6                   enable IPv6 support (disabled by default)
  --dbpath arg             directly access mongod database files in the given
                           path, instead of connecting to a mongod  server -
                           needs to lock the data directory, so cannot be used
                           if a mongod is currently accessing the same path
  --directoryperdb         if dbpath specified, each db is in a separate
                           directory
  -f [ --fields ] arg      comma separated list of field names e.g. -f name,age
  --fieldFile arg          file with fields names - 1 per line
  -q [ --query ] arg       query filter, as a JSON string
  --csv                    export to csv instead of json
  -o [ --out ] arg         output file; if not specified, stdout is used
  --jsonArray              output to a json array rather than one object per
                           line
```


3. mongodump

mongoimport 和 mongoexport 分别从集合中导入和导出数据，处理人可读的数据格式。如果要做热备，可以用 mongodump 以二进制格式转储整个数据库。以-help 为参数执行 mongodump 命令，可以查看 mongodump 的选项，输出如下：

> 不带参数执行 mongodump 会转储相关的 MongoDB 数据库，所以别用 mongoimport 和 mongoexport 的方式来浏览参数。

```
options:
  --help                   produce help message
  -v [ --verbose ]         be more verbose (include multiple times for more
                           verbosity e.g. -vvvvv)
  -h [ --host ] arg        mongo host to connect to ("left,right" for pairs)
  --port arg               server port. Can also use --host hostname:port
  -d [ --db ] arg          database to use
  -c [ --collection ] arg  collection to use (some commands)
  -u [ --username ] arg    username
  -p [ --password ] arg    password
  --ipv6                   enable IPv6 support (disabled by default)
  --dbpath arg             directly access mongod database files in the given
                           path, instead of connecting to a mongod  server -
                           needs to lock the data directory, so cannot be used
                           if a mongod is currently accessing the same path
  --directoryperdb         if dbpath specified, each db is in a separate
                           directory
  -o [ --out ] arg (=dump) output directory
  -q [ --query ] arg       json query
```

了解完文档数据库的灵活性和维护工具，现在可以进行下一个主题：列数据库。

7.2　面向列数据库中数据 schema 的演进

HBase 不是完全弱 schema 的，它支持比较宽松的 schema，主要表现为列族定义。列族相对静态，定义列的逻辑分组，而列定义则更加动态灵活。为了解释清楚，这儿我要重用并扩展第 3 章中首次介绍 HBase 时使用的例子，它与博文有关。如果需要了解细节，可以回顾第 3 章里关于 HBase 的一节。那个集合中的元素类似下面这样：

```
{
    "post" : {
        "title": "an interesting blog post",
        "author": "a blogger",
        "body": "interesting content",
    },
    "multimedia": {
        "header": header.png,
        "body": body.mpeg,
    },
}
```

或

```
{
    "post" : {
        "title": "yet an interesting blog post",
        "author": "another blogger",
        "body": "interesting content",
    },
    "multimedia": {
        "body-image": body_image.png,
        "body-video": body_video.mpeg,
    },
}
```

blogposts.txt

用 `bin/start-hbase.sh` 启动 HBase，并用 `bin/hbase shell` 连上服务器。然后按顺序运行下面的命令来创建表并生成数据：

```
create 'blogposts', 'post', 'multimedia'
put 'blogposts', 'post1', 'post:title', 'an interesting blog post'
put 'blogposts', 'post1', 'post:author', 'a blogger'
put 'blogposts', 'post1', 'post:body', 'interesting content'
put 'blogposts', 'post1', 'multimedia:header', 'header.png'
put 'blogposts', 'post1', 'multimedia:body', 'body.mpeg'
put 'blogposts', 'post2', 'post:title', 'yet an interesting blog post'
put 'blogposts', 'post2', 'post:title', 'yet another interesting blog post'
put 'blogposts', 'post2', 'post:author', 'another blogger'
put 'blogposts', 'post2', 'post:body', 'interesting content'
put 'blogposts', 'post2', 'multimedia:body-image', 'body_image.png'
put 'blogposts', 'post2', 'multimedia:body-video', 'body_video.mpeg'
```

blogposts.txt

数据库准备好以后，可以执行 get 查询：

```
get 'blogposts', 'post1'
```

blogposts.txt

输出类似这样：

```
COLUMN                        CELL
 multimedia:body              timestamp=1294717543345, value=body.mpeg
 multimedia:header            timestamp=1294717521136, value=header.png
 post:author                  timestamp=1294717483381, value=a blogger
 post:body                    timestamp=1294717502262, value=interesting content

 post:title                   timestamp=1294717467992, value=an interesting blog
 post
5 row(s) in 0.0140 seconds
```

现在数据集已经准备好了，我将重述一些 HBase 的基本概念，然后展示 HBase 是如何随着数据 schema 的变化而改变的。

首先，在 HBase 中，数据更新是覆写记录的新版本，而不是原地更新记录。在 CouchDB 中我们见过类似的行为。HBase 默认保存最新的三个版本，不过可以通过配置指定存储三个以上版本。版本数量在列族上设置，定义列族时可以指定版本的数量。在 HBase shell 里可以创建一个名为'mytable'的表，定义名为'afamily'的列族，指定保存 15 个版本：

```
create 'mytable', { NAME => 'afamily', VERSIONS => 15 }
```

VERSIONS 属性取整数值，所以它的最大值是 Integer.MAX_VALUE。尽管版本数量值能定义得非常大，但要用这个数据查询和获取值却不容易，因为没有内建的基于版本的索引。同样，版本还带有时间戳，但是在这个时间序列上查询数据集也不是可以轻易或高效实现的。

在这里我们使用命令行工具完成了配置，这个配置也可以通过编程实现。最大版本数需要作为参数传给 HColumnDescriptor 的构造函数。

在 HBase 中，列不需要提前定义好，因此给管理 schema 的变化带来了一定的灵活性。另一方面，列族相对静态。一个列族的列不能重命名，或者重分配到其他列族。实现这样的修改需要创建新列，从已有列迁移数据到新列，然后可能删除旧列。

HBase 支持用 shell 或编程方式创建列族，过去 Cassandra 在这方面更僵化一些。在旧版本的 Cassandra 里，定义列族要重启数据库。现在 Cassandra 灵活多了，允许在运行时修改配置。

7.3　HBase 数据导入与导出

表'blogposts'中的数据可以被导出到本地文件系统或者 HDFS 中。

要导出数据到本地文件系统，可以这样做：

```
bin/hbase org.apache.hadoop.hbase.mapreduce.Driver export blogposts
 path/to/local/filesystem
```

要导出同一份数据到 HDFS 中，则这么做：

```
bin/hbase org.apache.hadoop.hbase.mapreduce.Driver export blogposts
 hdfs://namenode/path/to/hdfs
```

除了导出，也可以导入数据到 HBase 表中。可以从本地文件系统或者 HDFS 导入数据。与导出类似，从本地文件系统导入数据命令如下：

```
bin/hbase org.apache.hadoop.hbase.mapreduce.Driver import blogposts
 path/to/local/filesystem
```

从 HDFS 导入与之类似，可以像下面这样导入：

```
bin/hbase org.apache.hadoop.hbase.mapreduce.Driver import blogposts
 hdfs://namenode/path/to/hdfs
```

7.4　键/值存储中的数据演变

键/值存储支持的数据集往往有限，要么为字符串，要么为对象值。有些键/值存储（比如 Redis）支持一些非常复杂的数据结构。还有些键/值存储，例如 Memcached 和 Membase，存储时间敏感的数据，并根据配置清除所有旧数据。

Redis 支持的集合类型有哈希、无序集、列表等，但几乎没有任何元数据设施。对 Redis 来说，所有东西都是哈希，或者哈希集合。它完全不知道键是什么，表示什么意思。

键/值存储并不保存文档、数据结构或对象，除了键/值对以外，对数据的 schema 几乎没有概念。所以 schema 的变化和键/值存储没有什么相关性。

和重命名字段类似的是重命名键。如果键存在，则可以像下面这样重命名之：

```
RENAME old_key_name new_key_name
```

Redis 通过将数据刷到磁盘上来完成它所持有的所有数据的持久化。要备份 Redis 数据库，只需复制 Redis 的 DB 文件，然后配置另一个实例来使用它就行。或者可以发送 `BGSAVE` 命令来异步启动并保存数据库持久化的任务。

7.5　小结

NoSQL 数据库支持弱 schema 结构，因此能适应灵活的持续的演进。尽管没有显式声明，但这实际上是 NoSQL 的一个关键特性。没有严格的 schema 使文档数据库可以专注于存储表达现实世界的数据，而无需把它们生硬地塞进规范化关系型模型里。

在列数据库中，去除严格的 schema 让维护工作和稀疏数据的增加都变得非常容易。在键/值存储中，schema 的概念则非常有限。

第 8 章

数据索引与排序

本章内容

- 创建索引以提高查询性能
- 在文档数据库和列族数据库中创建和维护索引
- NoSQL 数据排序
- 有效的设计决策：创建最优索引和排序模式

前面我们学习了 NoSQL 数据库的查询，这一章将学习如何确保查询快速高效。关系型数据库通常利用索引来优化查询的性能，类似的概念同样适用于 NoSQL。

索引的目的是为了提高数据访问性能。理论上，它们和书的索引在行为上类似。如果需要查找书里的术语或词语，有两个选择：

- 一页一页翻遍整本书查找术语或词语。
- 查看书后面的索引，找出包含术语或词语的页码，然后按页码浏览这些页。

两个选项相比，明显应该查索引而不是按页翻。索引让工作变得既简单又节省时间。

类似地，在访问数据库记录时，也有两个选择：

- 按条目一个一个地遍历整个集合或数据集。
- 利用索引快速获取到相关数据。

很明显，这一次索引又成了首选项。虽然书的索引和数据库的索引很类似，但是把这种相似性延伸太远又会造成困惑。书的索引用在文本上，因此索引词主要集中在比较重要的子集上。另一方面，数据库索引却可以应用到集合的所有数据上。索引通常建在条目的标识符或者特定属性上。

8.1 数据库索引的基本概念

创建索引没有万能公式，不过大部分有用的方法都依赖于少数共通的想法。这些想法则建立在哈希函数、B 树和 B+树的基础上。本节我们要详细了解这些概念，理解基础的理论。

哈希函数是精确定义的数学函数，可以把巨大的，往往是变长的，而且复杂的数据值变成单个整数或者一组字节码。哈希函数的输出有很多名字，包括哈希码、哈希值、哈希和、校验和。哈希码通常用作关联数组（又叫哈希映射表）的键。如果要把复杂的数据库属性值变成哈希码，以便创建索引，这时哈希函数就能派上用场了。

树这种数据结构将一组值分布到像树一样的结构里。值按层次组织，节点之间存在链接或指针。二叉树是最多只有两个子节点的树：一个在左边，另一个在右边。一个节点可以是父节点，这样它最多能有两个子节点；或者它可以是叶子节点，这样它就是链条中最后的节点。树结构的底部是根节点。图 8-1 可帮助理解二叉树数据结构。

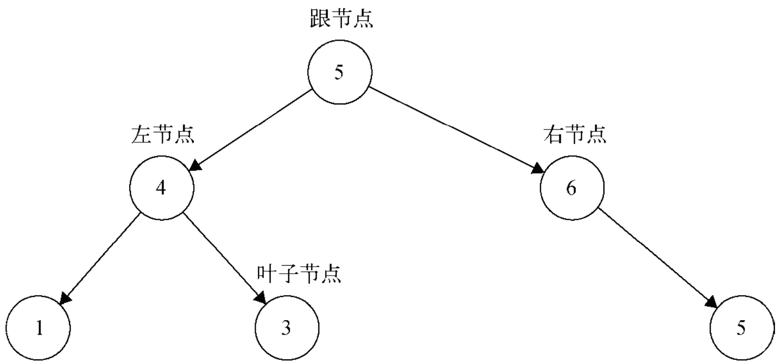


图　8-1

B 树是二叉树的泛化形式。一个父节点拥有的子节点数可以超过两个。B 树保持数据是有序的，进而支持高效查找和数据访问。B+树是 B 树的特例。在 B+树中，所有记录存在叶子中，叶子依次链接起来。B+树是数据库索引中最常使用的树结构。

如果想了解更多有关 B 树和 B+树的内容，可以阅读下列在线内容：

- http://en.wikipedia.org/wiki/B-tree
- www.semaphorecorp.com/btp/algo.html
- http://en.wikipedia.org/wiki/B%2Btree

另外，可以更加系统地阅读 Cormen、Leiserson、Rivest 和 Stein 合著的书 *Introduction to Algorithms*，ISBN 0-262-03384-4。

虽然基础模块一样，但是在不同的 NoSQL 产品里，创建和使用索引的方式各有不同。本章后续部分会介绍 MongoDB、CouchDB 和 Apache Cassandra 的索引。此外，在介绍索引的同时，还会介绍高效的数据排序，因为二者高度相关。

8.2　MongoDB 的索引与排序

MongoDB 提供了非常丰富广泛的索引选项以提高查询性能。默认情况下，它会在所包含的所有集合的`_id`属性上创建一个索引。

解释索引的最好方法是以例服人。我们先从第 6 章介绍的电影评分集合例子开始。如果你的 MongoDB 实例里没有电影评分集合，请按第 6 章的样例设置和加载集合。全都做完以后应该有三个集合，即`movies`、`ratings`和`users`。

为了解索引的意义和影响，还需要一些工具来测量有无索引时的查询性能。在 MongoDB 里，

使用内置工具就能方便完成测量，这些工具可以解释查询计划，并标记出执行缓慢的查询。

查询计划描述为了执行给定的查询请求，数据库必须要做的事情。在深入了解查询输出以及它所表达的内容前，先运行计划解释工具。要获取 ratings 集合中的所有条目，可以像下面这样查询：

```
db.ratings.find();
```

movielens_indexes.txt

要查看计划解释，执行下面这个查询：

```
db.ratings.find().explain();
```

movielens_indexes.txt

计划解释的输出类似下面这样：

```
{
    "cursor" : "BasicCursor",
    "nscanned" : 1000209,
    "nscannedObjects" : 1000209,
    "n" : 1000209,
    "millis" : 1549,
    "indexBounds" : {

    }
}
```

8

输出说总共消耗 1549 毫秒返回 1 000 209（超过 100 万）个文档。在返回的 1 000 209 个文档中，总共检查了 1 000 209 个文档，它还说使用了 BasicCursor。

很明显，解释函数的输出也是一个文档。如例子展示的那样，文档的属性包括下面这些。

- Cursor。游标用来返回查询结果集。游标有两种类型：基本游标和 B 树游标。基本游标意味着表扫描，B 树游标表明用到了索引。
- Nscanned。被扫描的实体总数。使用索引时，它对应索引实体的总数。
- nscannedObjects。扫描的文档总数。
- N。返回的文档总数。
- Millis。查询的耗时，单位为毫秒。
- indexBounds。表示查询匹配的索引范围的最小键和最大键。这个字段只有在使用到索引时才有关。

下面的例子查询 ratings 的子集。ratings 集合包含用户对电影的评分（分值从 1 到 5）。为了过滤 ratings 集合，将其限制到特定一部电影相关的子集。ratings 集合只包含电影的唯一标识符，所以为了把标识符和电影名关联起来，需要查找 movies 集合中的值。我选择影片 *Toy Story* 最初的版本（即 *Toy Story 1*）作为例子，你可以另外挑一部。

要获取与 *Toy Story* 有关的文档，可以利用正则表达式。前面在第 6 章里了解过这个查询过滤技术。如果不清楚，别犹豫，赶紧翻看那一章，回顾这些概念。对 movies 集合中所有与 *Toy Story*

有关的文档可以查询如下：

```
db.movies.find({title: /Toy Story/i});
```

movielens_indexes.txt

输出应该是这样：

```
{ "_id" : 1, "title" : "Toy Story (1995)", "genres" : [ "Animation",
"Children's", "Comedy" ] }
{ "_id" : 3114, "title" : "Toy Story 2 (1999)", "genres" : [ "Animation",
"Children's", "Comedy" ] }
```

我猜在汇编这些评分数据时，*Toy Story 3* 还没有发布。这就解释了为什么在列表里看不到它。下面用`"Toy Story"`的电影标识符（正好是 1）找出所有用户的相关评分。做之前，先运行计划解释函数，查看数据库如何执行正则表达式查询，以便从 `movies` 集合中找出 `Toy Story`。你可以运行计划解释函数如下：

```
db.movies.find({title: /Toy Story/i}).explain();
```

movielens_indexes.txt

输出应该是这样：

```
{
    "cursor" : "BasicCursor",
    "nscanned" : 3883,
    "nscannedObjects" : 3883,
    "n" : 2,
    "millis" : 6,
    "indexBounds" : {

    }
}
```

跑个计数，在 `movies` 集合上运行 `db.movies.count();`来确认文档数量，会发现它和查询解释里的 `nscanned` 及 `nscannedObjects` 值相对应。这说明正则表达式查询导致了表扫描，这样效率就差了。因为文档只有 3833 个，所以查询足够快，只消耗了 6 毫秒。很快你就会知道如何利用索引提高这个查询的效率，但现在先返回 `ratings` 集合，获取与 *Toy Story* 有关的子集。

要列出所有 `Toy Story`（更准确地说是 *Toy Story(1995)*）有关的评分，可以这样查询：

```
db.ratings.find({movie_id: 1});
```

movielens_indexes.txt

要查看这个查询的查询计划，运行 explain 如下：

```
db.ratings.find({movie_id: 1}).explain();
```

movielens_indexes.txt

输出应该是这样：

```
{
    "cursor" : "BasicCursor",
    "nscanned" : 1000209,
    "nscannedObjects" : 1000209,
    "n" : 2077,
    "millis" : 484,
    "indexBounds" : {

    }
}
```

到这一步就非常清楚了,查询显然未优化,因为 nscanned 和 nscannedObjects 都有 1 000 209 次读，包括了集合的所有文档。这样我们就正好开始介绍索引和优化。

8.3　MongoDB 里创建和使用索引

在 MongoDB 里，关键字 ensureIndex 完成了大部分创建索引的工作。上一节最后一个查询按 movie_id 过滤 ratings 集合，所以在这个属性上创建索引应该能把表扫描变成 B 树索引遍历。下面，先验证理论的正确性。

执行下面的命令创建索引：

```
db.ratings.ensureIndex({ movie_id:1 });
```

movielens_indexes.txt

这样就会在 movie_id 上创建一个索引，按升序排列索引中的键。要创建降序索引，可使用下面的命令：

```
db.ratings.ensureIndex({ movie_id:-1 });
```

movielens_indexes.txt

然后重新运行原来的查询：

```
db.ratings.find({movie_id: 1});
```

movielens_indexes.txt

然后再确认查询计划：

```
db.ratings.find({movie_id: 1}).explain();
```

movielens_indexes.txt

输出应该是这样：

```
{
    "cursor" : "BtreeCursor movie_id_1",
    "nscanned" : 2077,
    "nscannedObjects" : 2077,
    "n" : 2077,
```

```
        "millis" : 2,
        "indexBounds" : {
            "movie_id" : [
                [
                    1,
                    1
                ]
            ]
        }
}
```

乍看之下，很明显查找的条目（和文档）数量从 1 000 209（集合中的文档总数）减少到了 2077（匹配过滤条件的文档数量）。多么巨大的性能提升！从算法角度讲，文档搜索耗时已经由线性可扩展缩减为常量。因此，运行查询的总耗时从 484 毫秒缩减为 2 毫秒，减少了 99%。

从查询计划的游标值可以看出，很明显用到了索引 movie_id_1。可以试着创建一个按降序排列的索引，然后再重新执行查询和查询计划。不过在执行查询前，先分析一下集合 ratings 的索引列表，找出如何强制使用某个特定的索引。

要获取所有索引的列表（准备说是数组）非常容易。可以像下面这样查询：

```
db.ratings.getIndexes();
```

movielens_indexes.txt

在 movie_id 上有两个索引，分别为升序和降序，加上默认_id 上的索引，列表里应该总共有三个索引。getIndexes 的输出如下：

```
[
    {
        "name" : "_id_",
        "ns" : "mydb.ratings",
        "key" : {
            "_id" : 1
        }
    },
    {
        "_id" : ObjectId("4d02ef30e63c3e677005636f"),
        "ns" : "mydb.ratings",
        "key" : {
            "movie_id" : -1
        },
        "name" : "movie_id_-1"
    },
    {
        "_id" : ObjectId("4d032faee63c3e6770056370"),
        "ns" : "mydb.ratings",
        "key" : {
            "movie_id" : 1
        },
        "name" : "movie_id_1"
    }
]
```

前面已经用下面的命令在 movie_id 上创建了一个降序索引。

```
db.ratings.ensureIndex({ movie_id:-1 });
```

movielens_indexes.txt

如果需要，通过 hint 方法可以强制查询使用特定的索引。要强制使用 movie_id 上的降序索引来获取“Toy Story(1995)”，可以这样查询：

```
db.ratings.find({ movie_id:1 }).hint({ movie_id:-1 });
```

movielens_indexes.txt

完成执行以后，通过查询计划查看使用了哪个索引，表现如何。对上面使用 movie_id 上降序索引进行的查询，访问查询计划如下：

```
db.ratings.find({ movie_id:1 }).hint({ movie_id:-1 }).explain();
```

movielens_indexes.txt

查询计划输出如下：

```
{
    "cursor" : "BtreeCursor movie_id_-1",
    "nscanned" : 2077,
    "nscannedObjects" : 2077,
    "n" : 2077,
    "millis" : 17,
    "indexBounds" : {
        "movie_id" : [
            [
                1,
                1
            ]
        ]
    }
}
```

输出确认了对 movie_id 上标识为 movie_id_-1 的降序索引的使用。它还显示出降序索引和升序索引一样，也只访问了 2077 个条目。

不过输出中却有一个奇怪之处。尽管使用了索引，而且只选择一小部分文档进行扫描，但是返回结果集还是花了 17 毫秒。这个数字比表扫描所用的 484 毫秒少很多，但明显高于升序索引所消耗的 2 毫秒。在这个例子里很有可能是因为，movie_id 值为 1，并且位于升序列表的头部，而且前一个查询已经缓存了结果。访问列表头部的文档时，不能说升序索引的性能肯定会超过降序索引。同样，访问列表尾部的文档时，也不能说降序索引的性能肯定会超过升序索引。大部分情况下，特别是对列表中部的条目，两种索引的性能同样好。要验证这个性能断言，可以分别使用两个索引查找一部电影的评分，电影的 movie_id 位于列表另一端。

集合 ratings 的 movie_id 字段（或属性）对应 movies 集合的_id 字段。_id（以及

movie_id）字段取值为整数，所以按降序找出顶端的 movie_id 和找出 movies 集合中_id 字段的最大值一样。找出 movies 集合中_id 最大值的一种办法是像下面这样按降序排列：

```
db.movies.find().sort({ _id:-1 });
```

movielens_indexes.txt

JavaScript 命令行每次只返回 20 个文档，所以很容易一眼找出最大值是 3952。如果用编程语言 API 或任何其他机制运行这个查询，可能最好要限制结果条目的数量，因为只需要一个。可以像下面这样运行查询：

```
db.movies.find().sort({ _id:-1 }).limit(1);
```

movielens_indexes.txt

> 为什么在返回降序列表的头几个条目时，不用 findOne 方法而用 limit 方法？因为排序不支持 findOne。这是因为 findOne 可以只返回一个文档，而对一个文档进行排序没有任何意义。另一方面，limit 方法只限定最终输出为总结果集的一个子集。

movie_id 值 3952 对应影片 *The Contender(2000)*。要获取 *The Contender* 的评分，无论 movie_id 上的升序还是降序索引都可以。因为这里的目标是分析满足边界条件的查询性能，所以可以两个都用。两种情况下都执行查询计划。movie_id 上升序索引的查询和查询计划命令如下：

```
db.ratings.find({ movie_id:3952 }).hint({ movie_id:1 });
db.ratings.find({ movie_id:3952 }).hint({ movie_id:1 }).explain();
```

movielens_indexes.txt

查询计划的输出像这样：

```
{
    "cursor" : "BtreeCursor movie_id_1",
    "nscanned" : 388,
    "nscannedObjects" : 388,
    "n" : 388,
    "millis" : 2,
    "indexBounds" : {
        "movie_id" : [
            [
                3952,
                3952
            ]
        ]
    }
}
```

movie_id 上降序索引的查询和查询计划如下：

```
db.ratings.find({ movie_id:3952 }).hint({ movie_id:-1 });
db.ratings.find({ movie_id:3952 }).hint({ movie_id:-1 }).explain();
{
    "cursor" : "BtreeCursor movie_id_-1",
    "nscanned" : 388,
    "nscannedObjects" : 388,
    "n" : 388,
    "millis" : 0,
    "indexBounds" : {
        "movie_id" : [
            [
                3952,
                3952
            ]
        ]
    }
}
```

movielens_indexes.txt

从多次执行查询的结果来看，极端值并不受益于从某端开始的索引，这个理论看似是真的。不过，要牢记查询计划的输出并不恒定。每次运行都可以产生不同的结果。例如，值可以被缓存，这样一来即便重跑也不会触及底层数据结构。此外，对较小的数据集（正如 movies 集合的例子），这点区别微不足道，而像 I/O 滞后这样的开销对响应时间的影响则要大得多。一般情况下，尤其是对大型数据集来说，应该使用有利于查询的排序索引。

有时，在对集合大量修改后应当重建索引。要重建 ratings 集合的所有索引，可以执行下面的命令：

```
db.ratings.reIndex();
```

movielens_indexes.txt

也可以使用 runCommand 来重新索引：

```
db.runCommand({ reIndex:'ratings' });
```

movielens_indexes.txt

大多数情况下无需重建索引，除非集合大小发生了明显的变化，或者索引占用了大到不寻常的磁盘空间。

有时可能需要删除旧索引，再创建新索引，而不是重建旧索引。可以用 dropIndex 命令删除索引：

```
db.ratings.dropIndex({ movie_id:-1 });
```

movielens_indexes.txt

这个命令删除了 movie_id 上的降序索引。如果需要也可以删除所有索引。要删除（除_id 字段索引以外的）所有索引，可以这么做：

```
db.ratings.dropIndexes();
```

movielens_indexes.txt

8.3.1　组合与嵌套键

到目前为止，我们还只是在单个字段（或属性）上创建索引。其实可以创建包含多个字段的组合索引。例如，可以在 movie_id 和 rating 字段上一起创建索引。创建这个索引的命令如下：

```
db.ratings.ensureIndex({ movie_id:1, rating:-1 });
```

movielens_indexes.txt

这会在 movie_id（升序）和 rating（降序）上创建一个组合索引。除此之外，在 movie_id 和 rating 上四个可能的组合索引中还有三个可以创建。共有四种可能是因为两个键分别有升降两种排序。排序会影响涉及排序和范围的查询，所以在定义集合的组合索引时一定要注意排序。

包含 movie_id 和 rating 的组合索引可以用来查询同时匹配这些键或者只是第一个键（即 movie_id）的文档。如果只基于 movie_id 用这个索引过滤文档，则其行为与 movie_id 上的单字段索引类似。

组合索引包含的键不限于两个，可以包括任意多个键。可以像下面这样为 movie_id、rating 和 user_id 创建一个组合索引：

```
db.ratings.ensureIndex({ movie_id:1, rating:-1, user_id:1 });
```

movielens_indexes.txt

这个索引可以用于查询以下任何情况的组合：

- movie_id、rating 和 user_id
- movie_id 和 rating
- movie_id

组合索引也可以包含嵌套字段。在了解如何创建包含嵌套字段的组合索引前，我们先来介绍如何创建包含单个嵌套字段的索引。为了演示，使用一个有关人的集合（people2）样例。一个 people2 集合的元素示例如下：

> 我已经有了一个 people 集合，所以第二个叫 people2。你可以随意选择自己喜欢的名字。

```
{
    "_id" : ObjectId("4d0688c6851e434340b173b7"),
    "name" : "joe",
    "age" : 27,
    "address" : {
        "city" : "palo alto",
        "state" : "ca",
        "zip" : "94303",
        "country" : "us"
    }
}
```

可以在 address 字段的 zip 字段上创建一个索引，像这样：

```
db.people2.ensureIndex({ "address.zip":1 });
```

movielens_indexes.txt

还可以为 name 和 address.zip 字段创建一个组合索引：

```
db.people2.ensureIndex({ name:1, "address.zip":1 });
```

movielens_indexes.txt

也可以选择整个子文档作为索引键，就是用 address 字段创建一个索引：

```
db.people2.ensureIndex({ address:1 });
```

movielens_indexes.txt

这个索引包括整个文档，不只是文档的 zip 字段。如果传入整个文档来查询，就能用上这个索引。

MongoDB 集合的字段可以包含数组，而不是文档。这种字段也可以被索引。现在我们来查看另一个 orders 集合的例子，以演示如何索引数组字段。一个 orders 集合的元素示例如下：

```
{
    "_id" : ObjectId("4cccff35d3c7ab3d1941b103"),
    "order_date" : "Sat Oct 30 2010 22:30:12 GMT-0700 (PDT)",
    "line_items" : [
        {
            "item" : {
                "name" : "latte",
                "unit_price" : 4
            },
            "quantity" : 1
        },
        {
            "item" : {
                "name" : "cappuccino",
                "unit_price" : 4.25
            },
            "quantity" : 1
        },
```

```
        {
            "item" : {
                "name" : "regular",
                "unit_price" : 2
            },
            "quantity" : 2
        }
    ]
}
```

可以这样索引 line_items：

```
db.orders.ensureIndex({ line_items:1 });
```

movielens_indexes.txt

如果索引字段包含数组，那么数组的每个元素都会被添加到索引中。

此外，还可以按 line_items 数组的 item 属性索引：

```
db.orders.ensureIndex({ "line_items.item":1 });
```

movielens_indexes.txt

再进一个层次，按 line_items 数组的 item 文档的 name 属性索引：

```
db.orders.ensureIndex({ "line_items.item.name":1 });
```

movielens_indexes.txt

这样就可以按嵌套 name 字段查询：

```
db.orders.find({ "line_items.item.name":"latte" });
```

movielens_indexes.txt

运行查询计划，得知这个查询的游标值是 BtreeCursor line_items.item.name_1，表明使用了嵌套索引。

8.3.2　创建唯一索引和稀疏索引

现在，相信你已经完全被说服，MongoDB 确实提供了丰富的选项来索引文档，而且提供了高效的查询性能。除了提高查询性能以外，索引还可以用作施加约束。

可以通过显式声明来创建一个稀疏索引：

```
db.ratings.ensureIndex({ movie_id:1 }, { sparse:true });
```

movielens_indexes.txt

稀疏索引意味着索引字段无值的文档会被索引彻底忽略。这点有时可能是需要的，但是要注意，稀疏索引可能没有引用集合中的所有文档。

MongoDB 还支持创建唯一索引。可以在 movies 集合的 title 字段上创建一个唯一索引：

```
db.movies.ensureIndex({ title:1 }, { unique:true });
```

movielens_indexes.txt

集合 movies 中任意两条目的标题都不相同，如果有标题相同的情况出现，唯一索引就不会创建，除非显式声明除第一个条目以外，所有重复都可以丢弃。这样的显式声明如下：

```
db.movies.ensureIndex({ title:1 }, { unique:true, dropDups : true });
```

movielens_indexes.txt

如果集合中一些文档的索引字段没有值，就会在这些位置上插入 null 值。与稀疏索引不同，这些文档不会被跳过。同样，如果有两个文档索引字段都没有值，只有第一个会保存，剩下的都会被忽略。

8.3.3 基于关键字的搜索和多重键

到目前为止，已经介绍了很多有关 MongoDB 索引的内容，覆盖到了所有基础概念和大部分细节。在转到下一个文档数据库 CouchDB 以前，再展示最后一个例子。这个例子与基于正则表达式的文本搜索有关。本章前面部分里，搜索电影 *Toy Story* 对应的电影标识符使用了下面这个查询：

```
db.movies.find({title: /Toy Story/i});
```

当时也运行了查询计划，结果显示扫描所有 3883 个文档共耗时 6 毫秒。集合 movies 很小，因此表扫描的代价并不是很大。不过这个查询要是放在大型集合上运行，可能就会慢很多。

要提高查询性能，可以像下面这样创建一个索引：

```
db.movies.ensureIndex({ title:1 });
```

不过有些情况下，创建一个传统索引可能不够，特别是当你不希望依赖于正则表达式，但又需要全文检索时。前面我们见过对包含数组的字段进行索引。在这种情况下，MongoDB 会创建多重键：为数组中的每个唯一值创建一个键。例如可以保存一个博文集合，名为 blogposts，其中每个元素形似如下：

```
{
    "_id" : ObjectId("4d06bf4c851e434340b173c3"),
    "title" : "NoSQL Sessions at Silicon Valley Cloud Computing Meetup in January
 2011",
    "creation_date" : "2010-12-06",
    "tags" : [
        "amazon dynamo",
        "big data",
        "cassandra",
        "cloud",
        "couchdb",
        "google bigtable",
```

```
        "hbase",
        "memcached",
        "mongodb",
        "nosql",
        "redis",
        "web scale"
    ]
}
```

现在，可以很容易地在 tags 字段上创建一个多重键索引，像这样：

```
db.blogposts.ensureIndex({ tags:1 });
```

它看起来像其他索引一样，但是可以按 tags 的任何值来搜索：

```
db.blogposts.find({ tags:"nosql" });
```

这个特性可以用来构建基于关键词的完整检索。和 tags 一样，需要把关键词保存到数组中作为字段值。MongoDB 不能自动实现关键字提取，你需要自己构建这部分系统。

维护大数组，查询大量文档而且每个文档都包含大数组，这些都会拖累数据库的性能。要发现和提前修正一些慢查询，可以利用 MongoDB 的数据库分析器。实际上，分析器可以记录所有操作。

分析器有如下三个级别。

- 0：分析器关闭。
- 1：只记录慢操作（大于 100 毫秒）。
- 2：记录所有操作。

记录所有操作是设置分析器级别为 2：

```
db.setProfilingLevel(2);
```

分析器的日志以 MongoDB 集合形式提供，可以用下面的查询来查看日志：

```
db.system.profile.find();
```

一路看到这里，理论上你已经了解了几乎所有与 MongoDB 索引及排序有关的内容。以后访问集合数据时，可以使用这些工具来调优查询到最佳性能。

8.4　CouchDB 的索引与排序

前面我们见过 CouchDB 的 REST 风格查询机制。现在再深入了解一下，为了提高查询效率数据是如何索引的。和 MongoDB 不同，CouchDB 的索引功能是自动的，在数据被修改以后，第一次读取时触发。要更好地理解这点，先回顾一下 CouchDB 的数据访问机制。CouchDB 遵循 MapReduce 风格的数据操作。

map 函数根据集合中的数据生成键/值对，最终生成视图结果。第一次访问视图时，会基于数据创建一个 B 树索引。之后的访问就从 B 树返回数据，不再直接碰底层数据，即第一次查询以后的其他查询都会利用 B 树索引。

CouchDB里的B树索引

B 树索引能很好地支持大数据量。即便数据增长量很大，B 树高度仍然能保持个位数并支持快速数据访问。在 CouchDB 里，B 树的实现为多版本并发控制（MVCC）和只追加（append-only）设计做了专门的调整。

多版本并发控制使得多个读写操作可以同时发生，无需排他锁。和它相似的最简单的就是像 GitHub 这样的分布式版本控制系统。所有的写操作序列化，读不受写影响。CouchDB 有一个`_rev`属性，用来保存最近的修订值。像乐观锁一样，写和读基于`_rev`值进行协调。

因此客户端每次读取数据时的版本都是当时的最新版本。视图结果中的索引会随着文档的修改和删除进行更新。

> couchdb-lucene 项目（https://github.com/rnewson/couchdb-lucene）使用 Lucene 开源搜索引擎和 CouchDB 来提供全文搜索能力。

8.5 Apache Cassandra 的索引与排序

像 HBase 和 Hypertable 这样的列族数据库，默认基于行键进行排序和索引。在这些数据库里，对列值的索引（又称次级索引）通常不提供现成的。HBase 对次级索引有一点点支持。Hypertable 计划在 1.0 版时支持次级索引，那也要等到今年晚些时候了。

Apahce Cassandra 是面向列数据库和纯键/值存储的一个混合体。它集成了 Google Bigtable 与 Amazon Dynamo 的想法。与面向列数据库类似，Cassandra 默认支持基于行键的排序和索引。此外 Cassandra 还支持次级索引。

用一个简单例子来解释一下 Cassandra 对次级索引的支持。你可能还记得第 2 章那个 Cassandra 数据库，它包含`CarDataStore`键空间和`Cars`列族。我们就用这个例子来解释。

先用`bin`目录下的`cassandra`程序启动 Cassandara 服务器，然后用 CLI 连上 Cassandra，如下：

```
PS C:\applications\apache-cassandra-0.7.4> .\bin\cassandra-cli -host localhost
Starting Cassandra Client
Connected to: "Test Cluster" on localhost/9160
Welcome to cassandra CLI.

Type 'help;' or '?' for help. Type 'quit;' or 'exit;' to quit.
```

如果试过第 2 章的例子，那 CarDataStore 应该已经在本地数据库中了。如果没有，请阅读第 2 章，并按要求设立好键空间和列族。设置完成以后，使`CarDataStore`成为当前键空间：

```
[default@unknown] use CarDataStore;
Authenticated to keyspace: CarDataStore
```

用下面的命令确认先前添加到本地 Cassandra 的数据还在：

```
[default@CarDataStore] get Cars['Prius'];
=> (column=make, value=746f796f7461, timestamp=1301824068109000)
=> (column=model, value=70726975732033, timestamp=1301824129807000)
Returned 2 results.
```

列族 Cars 中包含两列：make 和 model。为了提高按值查询 make 列的效率，可以在那列上创建一个次级索引。由于列已存在，需要修改定义来添加索引。更新列族和列定义如下：

```
[default@CarDataStore] update column family Cars with comparator=UTF8Type
...     and column_metadata=[{column_name: make, validation_class: UTF8Type,
 index_type: KEYS},
...     {column_name: model, validation_class: UTF8Type}];
9f03d6cb-7923-11e0-aa26-e700f669bcfc
Waiting for schema agreement...
... schemas agree across the cluster
```

cassandra_secondary_index.txt

update 命令在 make 列上创建了一个索引，索引类型是 KEYS。Cassandra 定义的 KEYS 类型索引近似一个简单的哈希键/值对。

现在，查询所有 make 值为 toyota 的数据。类 SQL 语法如下：

```
[default@CarDataStore] get Cars where make = 'toyota';
-------------------
RowKey: Prius
=> (column=make, value=toyota, timestamp=1301824068109000)
=> (column=model, value=prius 3, timestamp=1301824129807000)
-------------------
RowKey: Corolla
=> (column=make, value=toyota, timestamp=1301824154174000)
=> (column=model, value=le, timestamp=1301824173253000)

2 Rows Returned.
```

cassandra_secondary_index.txt

再来试一个，这次用 model 值为 prius 3 作为条件过滤 Cars 数据：

```
 [default@CarDataStore] get Cars where model = 'prius 3';
No indexed columns present in index clause with operator EQ
```

cassandra_secondary_index.txt

按 make 过滤没问题，但是按 model 过滤失败了。这是因为 make 上有索引，而 model 上没有。我们再试一个结合了 make 和 model 的查询：

```
 [default@CarDataStore] get Cars where model = 'prius 3' and make = 'toyota';
-------------------
RowKey: Prius
=> (column=make, value=toyota, timestamp=1301824068109000)
=> (column=model, value=prius 3, timestamp=1301824129807000)

1 Row Returned.
```

cassandra_secondary_index.txt

索引又生效了，因为至少一个过滤条件包括了被索引的值。

这个例子不包括值为数的列，所以没法展示大于、小于等过滤条件。如果真想要利用这类基于不等式的查询，那就不太走运了。因为目前 KEYS 索引还没有能力支持范围查询。如果将来 Cassandra 引入了 B 树，或类似类型的索引，就有可能支持通过索引进行范围查询。简陋的 KEYS 索引不足以支持范围查询。

8.6　小结

本章我们深入探讨了对 MongoDB 文档及其字段进行索引的细节，并且了解了 CouchDB 里的自动视图索引。一个突出的主题是这两个数据库都支持索引，而且这些索引和关系型数据库的索引也算不上大相径庭。

我们还了解了一些特殊的功能，例如 MongoDB 中数组的多重键索引，还有 CouchDB 自上次读取以来修改的所有文档的自动索引。

除了文档数据库的索引，我们还了解了流行的列族数据库 Apache Cassandra 的索引能力。

第9章

事务和数据完整性的管理

本章内容

- ❑ 理解 ACID 事务的要领
- ❑ 了解事务在分布式系统中的应用
- ❑ 理解 Brewer 的 CAP 定理
- ❑ 了解 NoSQL 产品的事务支持

要理解 NoSQL 世界里的事务和数据完整性，最好的方式是首先在熟悉的 RDBMS 环境里重温这些概念。一旦建立起基本的事务概念、术语，再演示一些例子，就更容易理解，在大规模分布式环境里事务受到哪些挑战，同时这些地方也正是 NoSQL 大放异彩之处。

不是所有 NoSQL 产品在事务和数据完整性上都共享同样的观点。所以在解释完对大规模分布式系统的事务完整性广泛的、普遍的期望后，最好还能展示一下特定产品的实现方式，这正是本章表达主题的方式。

所以让我们现在就开始吧，先从 ACID 谈起。

9.1 RDBMS 和 ACID

ACID，分别代表原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability），它们已是数据库系统事务完整性最高级别的黄金标准。正如缩写所暗示的，ACID 包含下面这些内容：

- ❑ **原子性**。一个事务操作要么完全成功，要么完全失败。这两个状态之间的任何不一致都不可接受。能说明此属性的一个典型例子是从账户 A 转账到账户 B。如果需要转 100 美元，须从 A 帐号转出 100 美元，再向 B 账户转入 100 美元。逻辑上转账操作包括两个步骤：从 A 扣除和向 B 转入。原子性意味着，如果出于某种原因从 A 转出成功后操作失败，那么整个操作必须回滚，操作不会停留在不一致的状态中（钱已从 A 转出，但未转入 B）。
- ❑ **一致性**。一致性意味着如果违反了预定义的约束或规则，数据就不会被持久化。如果某个字段说它只接受整数值，那么它就不会接受浮点值，除非四舍五入到最近整数。一致性和原子性容易混淆。RDBMS 中一致性通常与唯一约束、数据类型验证和引用完整性相

关。在较大的应用场景中，一致性可能还会包含更复杂的数据规则，但是在这样的情况下，维护一致性的任务主要还是留给了应用程序。

- **隔离性**。隔离性与数据的并发访问有关。如果两个独立的进程或线程操作同一个数据集，它们有可能会影响到对方。根据不同的需求，两个进程或线程可以被隔离开来。举个例子，假设有两个进程 X 和 Y 修改字段 V 的值，其初始值为 V0。X 读取 V0 并且想将其修改为 V1，但是就在它完成更新前 Y 也读取了 V0 并更新为 V2。现在当 X 要写入 V1 时发现初始值已经更新了。如果不控制，X 会覆写 Y 刚写入的新值，这可能不是我们想要的。图 9-1 形象地描述了这个例子。隔离性确保能避免这种矛盾。不同的隔离级别和策略将在下一节阐释。

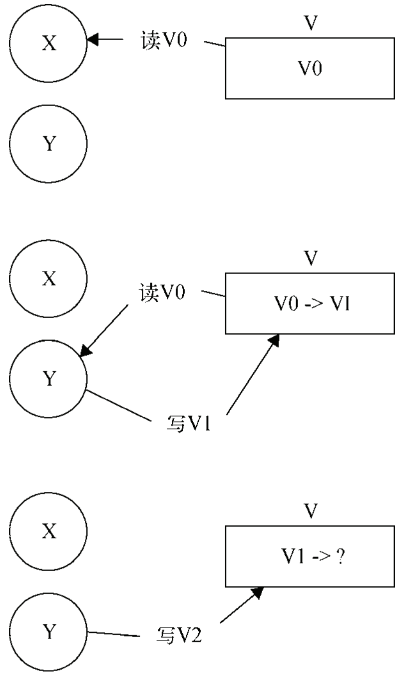


图 9-1

- **持久性**。持久性意味着一旦事务操作被确认了，它就一定会得到保证。使持久性遭受质疑的是下列情况：客户端程序收到了事务操作成功的确认，但是系统故障妨碍了数据被持久化到存储中。RDBMS 通常维护一个事务日志。事务只有在写入事务日志后才会被确认。如果在确认完成后、数据持久化前发生了系统故障，就会利用事务日志同步持久化存储，将其带回到一致状态中。

在 RDBMS 中，ACID 保证得到了广泛的认可。很多时候，使用 RDBMS 的应用程序框架和编程语言试图把 ACID 承诺扩展到整个应用中。如果整个栈（即数据库和应用）驻留在单台服务器或单个节点上，那当然没问题，但是一旦分布到多个节点上，它就开始捉襟见肘了。

隔离级别和隔离策略

严格的隔离级别会直接影响并发性，因此，为了允许并发处理就要放宽对隔离性的要求。ISO/ANSI SQL 标准定义了四个隔离级别，这四个级别分别提供渐进增强的隔离，它们是：

- 未授权读取
- 授权读取
- 可重复读取
- 可序列化

除此之外，无隔离级别或者说完全混乱可以看作第五级。隔离级别可以用例子解释清楚，所以在这里我会举个例子。假设存在一个简单的数据集合（或者 RDBMS 世界里的表），如表 9-1 所示。

表9-1　用于理解隔离级别的样例数据

标　　识	姓　　名	职　　业	所在地（城市）
1	James Joyce	Author	New York
2	Hari Krishna	Developer	San Francisco
3	Eric Chen	Entrepreneur	Boston

现在，假设有两个独立的事务：事务 1 和事务 2，并发操作这个数据集，顺序如下。

(1) 事务 1 读取集合中的所有三条数据。

(2) 事务 2 读取 ID 为 2 的记录，更新其 Location(City)属性"San Francisco" 为"San Jose"。不过它还没有提交修改。

(3) 事务 1 重读集合中所有三条记录。

(4) 事务 2 回滚步骤 2 中的所有更新。

随着隔离级别的不同，结果会有所不同。如果隔离级别是未授权读取，那么在第 3 步里，事务 1 会看见被事务 2 更新但未提交的修改（第 2 步）。到了第 4 步，这些未提交的修改会被回滚，这样的读被称为*脏读*。如果隔离级别更严格一些，设置为下一个级别：授权读取，那么第 3 步里事务 1 重读数据时就不会看到未提交的修改。

现在交换步骤 3 和 4，而且假设事务 2 提交了更新，那么新的步骤序列如下。

(1) 事务 1 读取集合中的所有 3 条数据。

(2) 事务 2 读取 ID 为 2 的记录并更新其 Location(City)属性"San Francisco"为"San Jose"，不过它还没有提交修改。

(3) 事务 2 提交第 2 步的更新。

(4) 事务 1 重读集合中的所有 3 条数据。

未授权读取级别不受修改步骤的影响。这个级别允许脏读，因此很明显读取已提交的更新不会遇到任何问题。授权读取就有所不同了。因为第 3 步里修改提交了，所以在第 4 步里事务 1 读取到了更新后的数据。从第 1 步和第 4 步读取到了不一样的值，所以这是一个不可重复读取

的例子。

当隔离级别提高到可重复读取时，第 1 步和第 4 步读取到的值就一样了。即两个事务并行处理时，事务 1 与事务 2 提交的更新被隔离开来。尽管在这个级别上可重复读是有保证的，但是插入和删除相关记录仍有可能发生。这会导致在一系列读取中，数据项时而包括在内，时而排除在外，这一现象称为*幻象读*。下面这个顺序的步骤演示了一个幻象读的例子。

(1) 事务 1 执行一个范围查询，请求 id 在 1 和 5 之间（包括 1 和 5）的所有数据项。因为集合里原来有三条记录且都满足条件，所以所有这三条记录都作为结果返回。

(2) 事务 2 接着插入一条新记录：`{Id = 4, Name = 'Jane Watson', Occupation = 'Chef', Location (City) = 'Atlanta'}`。

(3) 事务 2 提交步骤 2 中插入的数据。

(4) 事务 1 重新运行第 1 步的范围查询。

现在，随着隔离级别设置为可重复读取，步骤 1 和步骤 4 返回给事务 1 的数据集并不一样。在步骤 4 中，除原来的三条记录外，还能看到 id 为 4 的记录。为了避免幻象读，需要为读施加范围锁并使用最高级别的隔离：可序列化。*可序列化*意味着顺序处理，或者说串行的事务处理，但事实并非总是如此。在一些数据库里，可序列化隔离是通过快照来实现的。这些数据库在每个事务开始时为事务提供一个快照，然后只允许那些自创建快照以来没有发生任何改变的事务进行提交。

使用更高的隔离级别会增大饿死（starvation）和死锁的可能性。一个事务锁住了其他事务要使用的资源时会发生饿死，而两个并发事务相互等待对方释放资源时会发生死锁。

回顾完 ACID 事务和隔离级别的概念，现在可以开始探讨如何在高度分布式系统里运用这些想法了。

9.2 分布式 ACID 系统

要理解 ACID 是否适用于分布式系统，首先要了解分布式系统的各种属性，看看 ACID 对它们有哪些影响。

分布式系统有各种不同的形状、大小和形式，但是它们都具备几个典型特征，并且也面临相似的复杂性问题。随着分布式系统逐渐增大并伸展出去，复杂性的挑战更为突出。不止如此，如果系统需要提供高可用性，那么挑战会成倍增加。我们先从基本的情况开始，如图 9-2 所示。

两个应用程序，每个都连着一个数据库，所有四部分各自运行在单独的机器上。即便是这么简单的情况，提供 ACID 保证的挑战也非同小可。在分布式系统中，ACID 原则是在 open XA 协会所奠定理念的基础上使用的，它要求使用事务管理器或协调器来管理分布在多个事务资源上的事务。即使有中央协调器，实现跨数据库的隔离也极其困难。这是因为不同数据库提供的隔离保证不同。诸如两阶段锁（或其变种，强严格两阶段锁，SS2PL）和两阶段提交等一些技术有助于稍微改善情况。但是这些技术会导致操作阻塞，当事务还在运行，数据正在从一个一致性状态向另一个迁移时，还会阻碍部分系统可用。在长事务里，基于 XA 的分布式事务就不管用了，因为

长期阻塞资源访问是不切实际的。应用补偿操作等替代策略能帮助实现长时间分布式事务中的事务性保真度。

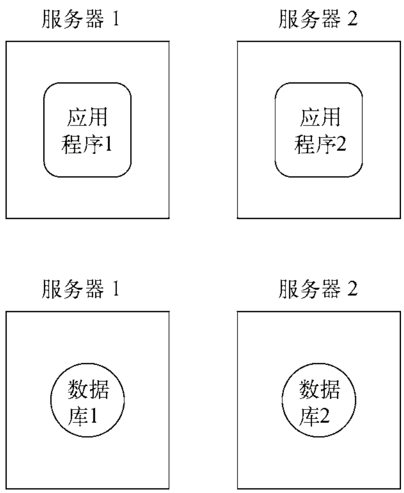


图 9-2

两阶段锁（Two-phase locking，2PL）是分布式事务锁的一种，锁只在第一阶段获取（而不释放），只在第二阶段释放（而不获取）。

SS2PL 是一种称为 commitment ordering 的技术的特例。更多相关内容请参阅 Yoav Raz (1992) 的研究论文"The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment"（www.vldb.org/conf/1992/P292.PDF）, Proceedings of the Eighteenth International Conference on Very Large Data Bases (VLDB), pp. 292-312, Vancouver, Canada, 1992 年 8 月, ISBN 1-55860-151-1 （还有 DEC-TR 841, Digital Equipment Corporation, 1990 年 11 月）。

两阶段提交（2PC）是这样一种技术，第一阶段里事务协调器和所有涉及的事务对象进行确认，而后第二阶段里对每个对象实际发送提交请求。这么做通常能避免部分失败，因为提交冲突在第一阶段里就可以被发现。

长事务中资源不可用的挑战在高可用场景里也会出现。对资源不可用或短缺的容忍度越小，这个问题就越突出。

要想一致地、有条理地解决在分布式系统中实现类 ACID 保证的难题，就需要理解下面三个因素在分布式系统中如何受影响。

- 一致性（Consistency）
- 可用性（Availability）

❑ 分区容忍性（Partition Tolerance）

一致性、可用性、分区容忍性（CAP）是 Brewer 定理的三大支柱，该定理承载着对这一代大型可扩展分布式系统事务完整性的思索。简单地说，Brewer 定理认为分布式（或者横向扩展）的系统不可能同时实现所有这三个特性（一致性、可用性、分区容忍性）。系统必须作出权衡，至少牺牲一样以成全其他两样。不过在讨论权衡之前，很重要的一点是首先了解这三个因素的含义。

9.2.1 一致性

一致性定义不是非常明确，在 CAP 的上下文里它意指原子性和隔离性。一致性意味着一致的读写，这样并发操作能看到同样的有效并且一致的数据状态，即至少没有过时的数据。

在 ACID 里，一致性（C）指没有满足约束的数据不会被持久化。这和 CAP 里的一致性不一样。

Brewer 定理是 Eric Brewer 推理出来的，2000 年作为 ACM Symposium on the Principles of Distributed Computing (PODC) 中的主题演讲由他亲自提出（www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf）。Brewer 有关 CAP 的想法是他在 UC Berkeley 和 Inktomi 工作的一部分。2002 年，Seth Gilbert 和 Nancy Lynch 证明了 Brewer 的推论，自此它被称为 Brewer 定理（有时称为 Brewer CAP 定理）。在 Gilbert 和 Lynch 的证明里，一致性被看作原子性。Gilbert 和 Lynch 的证明可以在已发表论文“Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”中找到。要获取这篇论文可以访问：http://theory.lcs.mit.edu/tds/papers/Gilbert/Brewer6.ps。

在单个节点上，一致性可以用数据库的 ACID 语义实现，但是当系统横向扩展出去，变成分布式以后，事情就变得复杂多了。

9.2.2 可用性

可用性意味着系统在需要时可以提供服务。作为推论，如果一个系统很忙、无法沟通，或者访问时没有响应，则被认为是不可用的。有些人，特别是那些试图反驳 CAP 定理及其重要性的人，争论说有轻微延迟或阻塞仍算得上可用。不过在 CAP 定义里没有任何含糊，只要系统在需要时不能服务请求，它就是不可用的。

这也就是说，许多应用程序都可以在可用性上做妥协，这是一个可选的权衡选项。

9.2.3 分区容忍性

并行处理和横向扩展已经被证明是行之有效的方法，而且正在逐渐被采纳为可扩展性和高性能计算的模型，与之对立的是向上扩展和构建大规模超级计算机。过去的几年已经表明，大多数情况下，建设一体化的巨型计算机既成本高昂又不切实际。添加大量商用硬件单元到集群中，使它们协同工作才是更具成本、算法和资源效率的高效解决方案。云计算的涌现正是这一事实的证明。

阅读下文的附注栏可以了解两种不同扩展策略之间的权衡。

因为选择了横向扩展的道路，就注定了分区和集群中偶尔会出现错误。CAP 的第三根支柱在于分区容忍性，或者说容错性。换句话说，分区容忍性衡量了系统在部分成员不可用情况下继续提供服务的能力。

垂直扩展的挑战与分布式计算的谬误

过去，传统选择更偏好一致性，因此一直以来系统架构师们会避开横向扩展，而选择向上扩展。向上扩展或者垂直扩展意味着需要更大更强的机器。很多情况下拥有更大更强的机器都没问题，但也常常具有以下特点。

- **厂商依赖**。并不是每个人都制造强大的机器，而强大机器的制造者们常常依赖专有技术，以提供你所需要的力量和效率。这就导致厂商依赖的可能性。厂商依赖本身并不是件坏事，至少不像它常被说成的那样。多年来，许多应用程序成功地构建出来，并运行在专有技术之上。然而它确实限制了你的选择，而且灵活性也比开放的对手更差。
- **更高成本**。强大的机器通常比一般商业硬件贵很多。
- **数据增长上限**。强大的机器能一直工作，直至数据增长到填满为止。到那时，除了换上更大的机器，或者横向扩展，没有其他选择。再大的机器，承载的数据量以及执行的处理数量也有限度。（现实生活中，团队比个人超级英雄更好！）
- **提前配置**。许多应用程序在刚开始的时候都不会意识到最终规模有多大。如果选择垂直扩展作为扩展策略，就需要为大规模提前做打算。预估和规划扩展性需求极其困难复杂，因为使用量、数据和交易的增长一般难以预测。

鉴于垂直扩展所带来的挑战，在近几年里水平扩展逐渐成为扩展策略之选。水平扩展意味着系统分布在多台计算机或节点上，其中每个节点可以是某种符合成本效益的商用机器。任何分布在多个节点上的事物都要受到分布式计算之谬的影响。下面是一个列表，其中列出了一系列在分布式系统背景下，开发者常认为是理所当然，但又常常不成立的假设。

- 网络是可靠的。
- 延迟为 0。
- 带宽是无限的。
- 网络是安全的。
- 拓扑不会发生变化。
- 只有一个管理员。
- 传输成本为 0。
- 网络都是同类型的。

分布式计算之谬归功于 Sum Microsystems（现在是 Oracle 的一部分）。Peter Deutsch 提出了列表中最初的七项。Bill Joy、Tom Lyon 和 James Gosling 也对这个列表作出了贡献。了解更多有关谬论的内容请访问：http://blogs.oracle.com/jag/resource/Fallacies.html。

9.3　维持 CAP

Brewer 定理指出，在一个大型分布式系统里同时实现一致性、可用性和分区容忍性是不可能的。你可以（而且应该）读读 Gilbert 和 Lynch 的证明，以深入理解如何以及为何 Brewer 是正确的。不过为了快速直观地演示，我用一个简单的例子来解释一下核心概念，这个例子用两个图显示：图 9-3 和图 9-4。

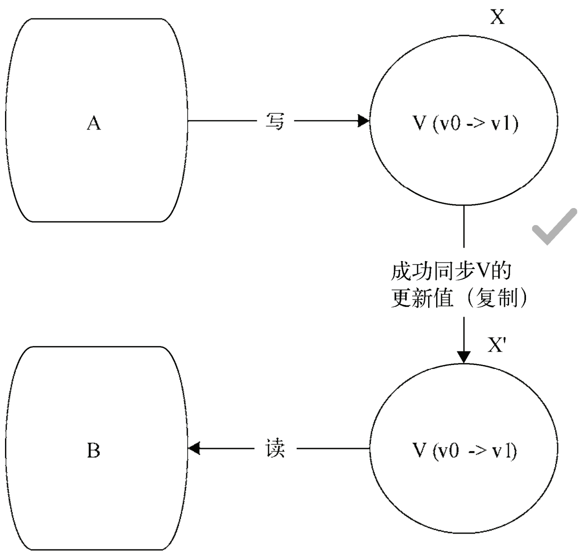


图 9-3

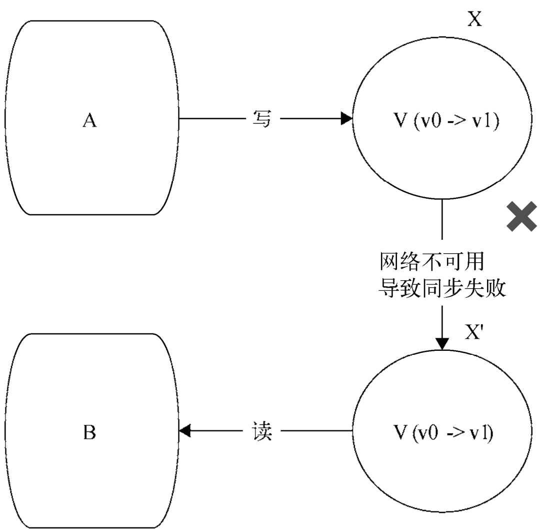


图　9-4

图 9-3 和 9-4 显示了一个集群系统的两个节点，进程 A 和 B 分别从 X 和 X'获取数据。X 和 X'是复制的数据存储（或结构），保存着相同数据的副本。A 写 X 而 B 读 X'。X 和 X'相互同步。V 是存储在 X 和 X'上的条目或对象。V 的初始值是 v0。图 9-3 展示了一个成功的例子，A 写 v1 到 V（更新其初始值 v0），v1 从 X 同步到 X'，然后 B 从 X'读取 V 值 v1。图 9-4 展示了一个失败的例子，A 写 v1 到 V 而 B 读 V 值，但是 X 和 X'之间的同步失败了，因此 B 读到的值与最新的 V 值并不一致。B 仍然读到 v0，但最新值是 v1。

如果要确保 B 总能读到正确的值，就需要确保从 X 同步复制 v1 到 X'。换句话说，以下这两个操作：(1)A 更新 X 中 V 值 v0 为 v1；(2)V 的更新值（即 v1）被从 X 复制到 X'，必须处于一个事务中，描述见图 9-5。这个设置会保证分布式事务的原子性，但会影响延迟和可用性。如果图 9-4 的失败出现，资源会被阻塞，直至网络恢复且从 X 到 X'的复制完成为止。

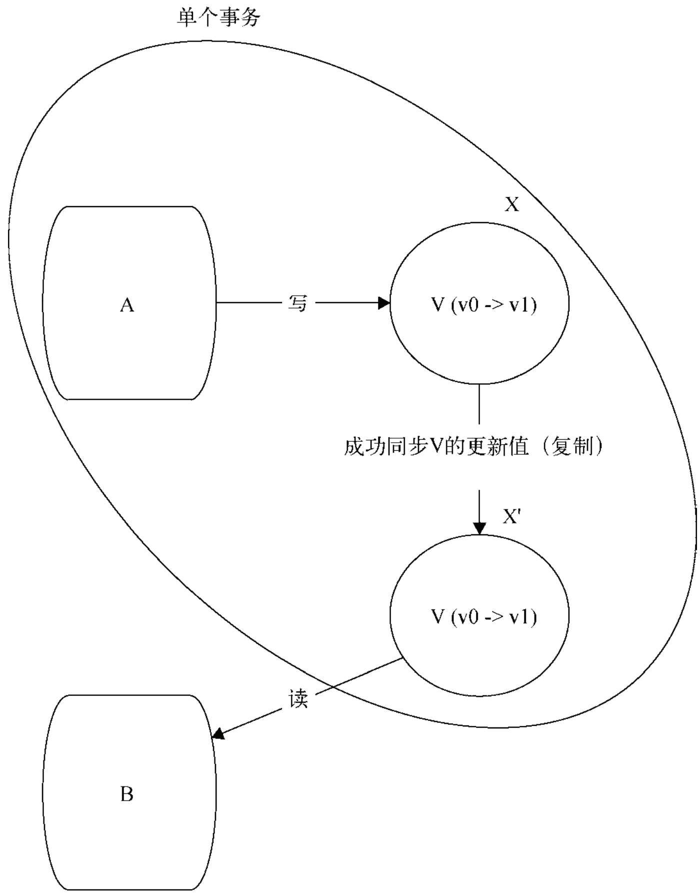


图　9-5

如果 X 和 X'之间的数据复制是异步的，那就没办法知道它发生的准确时间。如果不能知道事件发生的准确时间，很明显也无从保证事件发生了，除非显式寻找共识或确认。如果需要等待共识或确认，那异步操作对延迟或可用性的影响和同步操作就没太大区别。所以无论何种方式，只要系统是分布式的，且错误会发生，就需要明白数据一致性、系统可用性和分区容忍性之间的权衡，选择两项更重要的，并因此让第三项作出妥协。

下面是可能的选择。

- 选项 1：妥协可用性，选择一致性和分区容忍性。
- 选项 2：妥协分区容忍性，选择一致性和可用性。
- 选项 3：妥协一致性，但系统总是可用的，而且不受其中一部分被分区的影响。

传统 RDBMS 在横向扩展的情况下选择了选项 1。这种情况下，可用性会受到许多因素的影响，其中包括下面这些。

- 网络延迟造成延迟。
- 通信瓶颈。
- 资源不足。
- 硬件错误引发分区。

9.3.1 妥协可用性

极端情况下，节点失败后整个系统可能会完全不可用，直至节点恢复并且系统恢复到一致的状态。虽然不可用看似非常不利于业务的延续性，但有时这是唯一的选择。要么数据处于一致状态，要么事务失败。这是典型的金钱和时间敏感事务，失败情况下的补偿事务是完全无法接受的，或者要承担非常高的成本。两个账户转账的经典案例常作为这类用例被引用。不过在现实生活中，对这种极端情况，银行有时也会采用一些宽松的替代方案，稍后讨论弱一致性时会提到。

很多情况下，系统（包括那些基于 RDBMS 的）支持备份、快速复制和错误恢复。这意味着系统在很短的一段时间里可能还会处于不可用状态。大部分情况下，轻微的不可用并不是灾难性的，因此这是一个可行的选择。

9.3.2 妥协分区容忍性

有些情况下，最好还是不要妥协分区容忍性。刚刚我提到过，在横向扩展的系统中，节点失败是必然的，失败几率会随着节点数量的增加而增大。这样一来分区容忍性怎能是一个可选项？许多人会混淆分区容忍性和容错，但这两个概念并不相同。如果一个系统无法为受网络隔离的分区提供服务，但是能瞬间切换到其他节点，那它是容错的，但是不具备分区容忍性。

Google Bigtable 提供了数据存储的高可用性、强一致性，但在分区容忍性上做了妥协。这个系统是容错的，单节点失败时很容易存活，但它不是分区容忍的。更准确地说，在出错的情况下，它会标识出一个分区的主从部分，并尝试通过选举来解决问题。

为了进一步理解这点，可能要回顾在第 4 章里学的有关 Bigtable（及 HBase 等类似产品）的

内容。4.2 节的内容与此最相关。

Bigtable 以及相似的其他软件都使用了主从模式，其中列族数据存储在一个 region server 上。region server 由主服务器动态配置管理。在 Google Bigtable 中，数据存储在底层的 Google FileSystem（GFS）中，而整个架构通过 Chubby 进行协调，Chubby 使用像 Paxos 这样的选举算法来确保一致性。对 HBase，HDFS 执行与 GFS 相同的功能，而 ZooKeeper 替代 Chubby。ZooKeeper 使用选举算法从一个失败节点中恢复。

出现错误时，ZooKeeper 确定出哪个是主分区，哪个是从分区。基于这些推断，ZooKeeper 会将所有读写操作定向到主分区上，从分区改为只读，直至问题解决。

除此之外，Bigtable 和 HBase（及其底层文件系统，GFS 和 HDFS）对每个数据集分别存储三份。这样在三份中的一份出错或无法同步的情况下，可通过协商方式保持一致性。少于三份拷贝就无法达成共识。

9.3.3　妥协一致性

有些情况下，不能妥协可用性，而系统是如此高度分布，以至于分区容忍是必须的。这时就有可能要妥协强一致性。与强一致性对应的是弱一致性，因此所有这些妥协一致性的情况都可以算作是这一组。弱一致性是一个系列，包括没有一致性和最终一致性。对那些不限制更新数据方式的重要系统，数据不一致也许不可行，但最终一致性却可以。最终一致性的定义也不是很清晰，不过间接提到一个事实：经过更新，集群所有节点最终将看到同样的状态。如果“最终”可以限制在特定范围里，那么最终一致性模型就可行。

例如，即便无法与库存系统确认商品是否足够，购物车也可以允许下订单。少数情况下，订购的产品可能脱销了。这时，用户订单可以作为后台订单，进货时再处理。再举另一个例子，即便无法确认可用金额，银行卡也可以允许客户提取达到透支额度的金额，所以在最坏的情况下，即使钱不够，交易也仍然有效，可以用透支额度的方法。

要理解最终一致性，我们可以试着用下面三个条件来演示。

- R：被读取的节点的数量。
- W：被写入的节点的数量。
- N：集群节点总数。

这三个参数的不同组合对整体的一致性会有不同的影响。保持 $R < N$ 和 $W < N$ 可以允许高可用性。下面是一些值得了解的常见情况。

- $R+W>N$。这种情况下，很容易建立一致的状态，因为一些读和写的节点存在重叠。极端情况是：$R=N$ 且 $W=N$（即 $R+W=2N$），此时系统绝对是一致的，而且可以提供 ACID 保证。
- $R=1$，$W=N$。系统中读多于写时，把读负载分散到集群的所有节点上就是有意义的。$R=1$ 时，就读操作而言，每个节点都独立于其他节点。$W=N$ 的写配置意味着每次更新都会写所有节点。一旦某个节点失败，则整个系统都不可写。

- R=N，W=1。如果写 1 个节点就够，数据不一致的可能性就会非常高。不过如果 R=N，只有集群所有节点都可用时才能读。
- R=W=ceiling((N+1) / 2)。这种情况下能通过有效的选举提供最终一致性。

Eric Brewer 和他的同事们发明了 BASE 这个术语来表示最终一致性。BASE 表示基本可用软状态最终一致（Basically Available Soft-state Eventually consistent），它设计出来显然是要与 ACID 对立。不过 ACID 和 BASE 并不是对立的，实际上，它们不过是描述一致性光谱中的不同点罢了。

最终一致性表现为多种形式，也可以用很多方式来实现。一个策略是用面向消息的系统，另一个则利用基于选举的共识。在面向消息的系统里，可以使用消息队列传播更新。最简单的情况，可以用唯一的序列标识符对更新排序。第 4 章解释过 Amazon Dynamo 及其最终一致性模型的基本原理，你可以回去查阅。

下一节里，我会解释一些流行 NoSQL 产品中的一致性。我不会详尽地覆盖所有产品，而只会有选择地挑一部分讲。Google Bigtable 和 HBase 一致性模型已经介绍过了，所以我会跳过这些产品。本节将介绍文档数据库和最终一致性旗手（如 Cassandra）的一致性。

9.4 NoSQL 产品的一致性实现

本节先介绍分布式文档数据库 MongoDB 和 CouchDB 中的一致性。

9.4.1 MongoDB的分布一致性

MongoDB 没有规定一致性模型，但是默认强一致性。在某些情况下，MongoDB 可以，而且会配置为最终一致性。

在自动分片的、启用复制的集群中，默认每个分片里有一个主机。这种部署的一致性模型很强。不过某些情况下，可以部署 MongoDB 以提供更高的可用性和分区容忍性。其中一种情况是采用单个主机作为唯一的写节点，多个从节点读取。如果一个从节点已经从集群中分离出去，但还在为客户端提供服务，它可能会提供过时的数据。分区恢复以后，这个从节点会接收所有更新，从而实现最终一致。

9.4.2 CouchDB的最终一致性

CouchDB 的最终一致性模型依赖两个重要的属性：

- 多版本并发控制（MVCC）
- 复制

CouchDB 里每个文档都有版本，所有对文档的更新都会被标记上唯一修订号。CouchDB 是高可用的分布式系统，通过放宽一致性支持可用性。

发生读操作时，客户端（A）访问一个版本化的文档及其当前版本号。为了清楚起见，我们假设文档名为 D，当前版本或修订号是 v1。在客户端 A 忙着读取文档而且可能试图更新文档时，

客户端 B 访问了同一个文档 D，并了解到它的最新版本是 v1。客户端 B 位于能控制 D 的独立线程中。接下来，在客户端 A 返回前，客户端 B 做好了更新文档的准备，它更新了文档 D，并增加它的版本（或修订）号到 v2。当客户端 A 带着对文档 D 的更新回来时，它意识到文档自它上次访问已经被更新了。

这会在提交时导致冲突。幸运的是，版本（或修订）号可以用来解决这个冲突。图 9-6 是对这个更新冲突的图形展示。

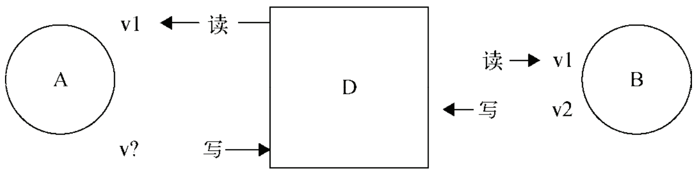


图　9-6

这样的冲突通常可以通过客户端 A 在提交更新前重读 D 来解决。在重读时，如果 A 发现它所使用的快照版本（这里是 v1）已经过期了，它就可以在最新读取的结果上再应用更新，然后提交更新。这种冲突解决方法在版本控制软件中很常见。现在许多版本控制软件（比如 Git 和 Mercurial）都采用 MVCC 来管理数据失真，避免提交冲突。

CouchDB 是可扩展的分布式数据库，所以尽管 MVCC 能处理单个实例上冲突的情景，但并不代表它能保持数据库的所有副本为当前最新。此时就是复制开始接手的时候了。复制作为一种常见的成熟技术，用于同步任意两个数据存储节点。它最简单的形式即文件同步程序 `rsync`，它能为文件系统单元（例如目录）实现同样的目的。

CouchDB 集群所有节点的数据会在复制进程的帮助下最终达到一致。在 CouchDB 中复制是增量的和容错的。因此，复制时只有修改（或增量更新）会被传送，传送过程中如果进程出错了，也可以平滑地恢复。CouchDB 的复制机制能感知状态，失败后会从上次停止的地方再重新开始。因此避免了多余的重启，并且设计中也考虑到了网络失败的内在倾向或节点不可用的情形。

CouchDB 的最终一致性模型既有效又高效。CouchDB 集群通常构成主-主节点，这样每个节点就可以独立服务请求，从而同时增强可用性和响应性。

综览文档数据库之后，让我们继续去了解一下 Apache Cassandra 的最终一致性模型。

9.4.3　Apache Cassandra的最终一致性

Apache Cassandra 的目标是成为像 Google Bigtable 和 Amazon Dynamo 那样的系统。从 CAP 定理的观点来看，这意味着 Cassandra 要提供两种权衡的选择。

- **支持一致性和可用性**：Bigtable 模型。
- **支持可用性和分区容忍性**：Dynamo 模型。

这两个模型本章前面都介绍过了。Cassandra 通过把最终的一致性配置留给开发者实现了这一点。作为一名开发者，你的选择如下。

- 设置 R + W > N 实现一致性。其中 R、W 和 N 分别代表读复制节点数量、写复制数量以及节点总数量。
- 设置 R = W= ceiling((N+1) / 2)实现选举。这是最终一致性的情况。

也可以设置写全一致的情况（其中 W = N），不过这种配置可能有些棘手，因为一旦失败，会导致整个应用不可用。

我们最后来快速了解一下 Membase 的一致性模型。

9.4.4　Membase的一致性

Membase 是兼容 Memcached 协议的分布式键/值存储，提供高可用性和强一致性，但不支持分区容忍性。Membase 是立即一致的。如果需要分区，可以用外部工具从 Membase 主机向从机复制数据，但这不是系统功能。

此外，Membase 和 Memcached 一样，善于保持时间敏感的缓存。在一个即刻的强一致性模型里，清除超过定义好的时间间隔的数据非常容易，而且有可靠的支持。为时间敏感数据支持不一致的窗口反而是个挑战。

在介绍过 NoSQL 的事务管理和一些产品之后，我们来对本章做个总结。

9.5　小结

尽管本章比较简短，但可能是本书中最重要的章节之一。本章清楚解释了 ACID 的概念及其可能的替代物 BASE，解释了 Brewer 的 CAP 定理，并试图将其重要意义与分布式系统关联起来，因为在流行的和广泛使用的各种系统中，它们正在逐渐成为标准。

一致性及其各种形式，不管是强的还是弱的，我们都分析了。最终一致性作为在分区条件下能提供更高可用性的一种可行的替代方案被提了出来。

由于 NoSQL 宽松的一致性配置，强一致性倡导者们一直都拒绝认真地考虑 NoSQL 数据库。虽然在许多交易系统中一致性都是重要需求，但是要么强一致要么没一致的方式也在用户中引起了许多的恐惧、不确定和怀疑。希望本章明确列出的这些选择能够让你有所理解。

最后，尽管本章解释了最终一致性，但并不是说建议采用它作为一致性模型。最终一致性有它的用武之地，在有分区且能安全提供高可用性的场景中应该使用最终一致性。不过要牢记，最终一致性也充满了潜在的麻烦。在最终一致性模型上设计和架构应用绝不是件轻松的事。如果事务完整性非常重要，缺少它会严重扰乱正常操作，那么就需要非常谨慎地决定是否采用最终一致性，要充分认识到所作选择的利弊。

Part 3

第三部分

熟悉 NoSQL

本部分内容

第10章 使用云中的 NoSQL

本章内容

- ❑ 探索随时可用的云中 NoSQL 数据库
- ❑ 利用 Google AppEngine 及其可扩展的数据存储
- ❑ 使用 Amazon SimpleDB

现在大部分的流行应用，比如 Google 和 Amazon，都通过水平扩展部署在大量机器和多个数据中心里，来实现高可用性和并发服务数百万用户的能力。像 Google 和 Amazon 这样的大规模 Web 应用的成功证明，在水平扩展环境中，NoSQL 解决方法往往比关系型对手更耀眼。水平扩展环境按需供应已被命名为“云”。如果可扩展性和可用性优先级更高，那么云中 NoSQL 可能就是比较理想的配置。

> 某些情况下，关系型和非关系型存储会组合使用。要说在水平扩展环境里只有 NoSQL 可行可能有失偏颇。应用程序的很多方面跟期望的规模、底层数据结构和事务完整性有关。

世界上存在许多云服务提供商和可用的 NoSQL 产品。就其中一些提供商来说，例如 Amazon EC2（Elastic Compute Cloud，弹性计算云），你可以选择安装任何想要使用的 NoSQL 产品。附录 A 包括了在 EC2 集群上安装一些流行 NoSQL 产品的指南。虽然有自己选择的自由，但也有一些云服务提供商提供了完全安装好、配置好的数据库基础设施，随时供你使用，这会让你省力不少。本章覆盖了云中 NoSQL 的选择。

云里的关系型数据库

许多关系型数据库都在云中提供，其中最突出的是下面这些。

- ❑ Windows Azure 平台上的 Microsoft SQL data service（microsoft.com/windowsazure）。
- ❑ Amazon Relational Database Service（RDS），它是 MySQL 实例的集群（http://aws.amazon.com/rds/）。

此外，还有许多 Amazon Machine Image（AMI）选项，支持 Oracle、PostgresSQL、MySQL 等，让你可以在 EC2 环境中上建立自己的数据库集群。一些 RDBMS 厂商（如 Oracle 和 Greenplum）也开始为私有云环境提供解决方案。尽管可能具备可扩展性，但在私有云是否算是云这个问题上，仍然存在公开的辩论。

本章介绍两种云中 NoSQL：Google Bigtable 和 Amazon SimpleDB，还会简短介绍其他一些新兴数据库：CouchOne、MongoHQ 和 Riak on Joyent's Smart machines。

Google 通过发布可对外提供服务的、易于使用的基础架构颠覆了整个云计算领域，但是 Google 并不是第一个发布云产品的。在 Google 首次宣布其服务时，Amazon EC2 早已是市场上声名显赫的一员了。然而 Google 的模型如此便利，它的云平台 Google App Engine（GAE）在很短的时间内就被广泛传播和采用。这个应用程序引擎也并非没有局限性，它的沙盒环境以及缺少对长时间运行进程的支持是令人讨厌的几方面之一。

本章从 GAE 基于 Bigtable 的数据存储开始，用插图和例子来展示这个数据存储的能力和建议的使用模式。

10.1 Google App Engine

Google App Engine（GAE）为应用程序提供了沙箱部署环境，应用程序可以用 Python 或者任意一种能够在 Java 虚拟机（Java Virtual Machine，JVM）上运行的语言来编写。Google 为开发者提供了丰富的 API 和 SDK，用来为应用服务引擎构建应用程序。

为了解释数据存储的功能和可用的数据建模 API，我们首先使用应用引擎的 Python SDK 来介绍相关内容。随后讨论超出共同概念的部分，特别是和 Java SDK 有关的功能。

10.1.1 GAE Python SDK：安装、设置和起步

首先需要安装 Python 和 GAE Python SDK。你可以从 python.org 下载 Python，从地址 http://code.google.com/appengine/downloads.html#Google_App_Engine_SDK_for_Python 得到 GAE Python SDK。详细的安装指南超出了本书的范畴，不过在所有支持的环境里安装 Python 和 GAE Python SDK 都相对简单直接。如果你在设置环境时遇到了问题，可以 Google 一下解决办法，和大多数开发者一样，你不会失望的。

虽然本章主要专注于 GAE 数据存储，不过了解如何在应用引擎上开发程序还是有好处的。针对 Python SDK，不妨稍微花点时间阅读一下名为“Getting Started: Python”的入门教程，在线访问地址：http://code.google.com/appengine/docs/python/gettingstarted/。在 GAE 上构建的应用都是 Web 应用。入门教程将会解释下列内容。

- 在 GAE 上构建 Python Web 应用的基础知识。
- 请求是如何处理的，响应是如何服务的。
- URL 是如何映射到处理程序上的。
- 如何包含动态和静态内容。
- 数据是如何建模和存储在底层数据存储中的。
- 如何使用模板来解耦视图和逻辑。
- 如何利用像认证、邮件、任务队列和 Memcache 这样的服务。
- 构建好的应用程序如何部署到本地开发服务器上。

❑ 如何把应用程序部署到生产环境里。

教程非常简洁准确，通过阅读它你可以快速上手进行基础开发。如果你用 Python 开发 Web 应用的经验有限，或者没有经验，那最好在学习本章前先学习一些基本的 Python Web 开发课程。如果你很熟悉 Python Web 开发，还是建议你快速浏览一下入门教程，以确保了解哪些工具和方法能用，哪儿需要使用其他策略。

> 如果你完全是新人，没有 Python 编程经验，建议阅读 Mark Pilgrim 的 *Dive into Python* 一书来学习语言基础，在线阅读可以访问：diveintopython.org。

接下来几节将深入了解数据建模和在 GAE 数据存储上对应用程序数据进行增删改查。为了生动具体一些，我们通过样例程序而不是抽象概念来讲解相关内容。

任务管理器：样例程序

我们来考虑一个简单的任务管理程序，用户可以定义任务、跟踪状态，并标记完成状态。要定义一个任务，用户需要提供名称和描述。可以添加标签进行分类，可以声明期望的到期时间。一旦完成，可以记录结束时间。任务属于用户，在应用程序的第 1 版里，除了拥有者以外，不和其他人分享任务。

要对任务进行建模，最好能罗列出所有属性，指明每个属性的数据类型，说明它是必需的还是可选的，是单值还是多值的。表 10-1 列出了任务的属性及其特征。

表　10-1

属性名	数据类型	是否必须	单值或多值
`Name`	String	Yes	Single
`Description`	String	No	Single
`start_date`	Date	Yes	Single
`due_date`	Date	No	Single
`end_date`	Date	No	Single
`Tags`	array (list collection)	No	Multiple

GAE Python SDK 提供的数据建模 API 支持开发者创建 Python 类型来表示任务。表示任务的模型类的最简形式可以是这样的：

```
import datetime
from google.appengine.ext import db

class Task(db.Model):
  name = db.StringProperty()
  description = db.StringProperty()
  start_date = db.DateProperty()
  due_date = db.DateProperty()
  end_date = db.DateProperty()
  tags = db.StringListProperty()
```

taskmanager GAE project

如果你用过 Django（djangoproject.com/）这样的 Web 框架，或者 SQLAlchemy（一种流行的 Python 数据库工具，sqlalchemy.org/）等 ORM 工具，就肯定见过类似的数据建模 API。GAE Python 数据建模 API 遵循 Python Web 开发者熟悉的语法和语义。

> ORM，或者对象–关系映射，提供了面对对象编程与关系型数据库世界之间的桥梁。

表 10-1 中，name 和 start_date 被声明为必需字段，但它们还没有纳入模型中。现在，修改 Task 类加入约束：

```
import datetime
from google.appengine.ext import db

class Task(db.Model):
  name = db.StringProperty(required=True)
  description = db.StringProperty()
  start_date = db.DateProperty(required=True)
  due_date = db.DateProperty()
  end_date = db.DateProperty()
  tags = db.StringListProperty()
```

taskmanager GAE project

> 有很多可用的验证选项。例如，required=True 强制字段必须存在。参数 choices=set(["choice1","choice2", "choice3", "choice4"]) 限制值必须是给定集合中的成员。自定义验证逻辑可以定义成函数作为参数值传入属性类的 validator 参数。

GAE 采用 Google Bigtable 作为数据存储。Bigtable 是一个有序的、分布式的、稀疏的、面向列族的映射表，它对列族里列的数量或类型以及存储在这些列中值的数据类型几乎没有限制。此外，Bigtable 还支持高效地存储稀疏数据集，允许一个表中两行数据拥有完全不同的列组合。它还允许相同列有不同的数据类型。换句话说，在一个数据存储中，两个同类（如 Task）实体可以有不同的属性集，或者两个同类实体可以有一个属性（相同名称）包含不同类型的数据。

数据建模 API 在包容性极强的 Bigtable 之上提供一层结构，并对属性的数据类型、值的集合和它们之间的关系提供了应用层面的限制。在描述“任务”实体的例子中，名为 Task 的 Python 类定义了任务的数据模型。

GAE 数据存储可以被看作对象存储，其中每个实体都是一个对象。这意味着实体或成员可以是 Python 类（例如 Task）的实例。类名 Task 是实体的类型。由键唯一标识实体，使其与数据存储中的其他实体区分开来。键可以是组合的标识符，包含下列元素。

- 继承路径

- 实体类型
- 实体 ID 或实体键名

如果 BaseTask 类型的实体是 Task 实体的父实体，那么 Task 实体的继承路径就包括了对 BaseTask 类型父实体的引用。Task 本身是实体类型。Task 类型的实体包含 ID，可以被看作主键。ID 可以是下列任意一个。

- 应用程序提供的值，名为 key_name，为字符串类型。
- 系统（比如 GAE）生成的唯一的数字 ID。

可以创建和保存实体如下：

```
task = Task(name = "Design task manager app",
            description = "Design the task management application.
Create the initial blueprint and the app architecture.",
            start_date = datetime.datetime.now().date())
task.put()
```

taskmanager GAE project

上面的代码创建了一个任务实例，它通过传值给构造函数的 name、description 和 start_date 参数创建而成。或者也可以先创建实例，再分别给每个属性赋值。在初始化时一定要传入所有必需属性的值。非必需属性的值可以通过两种途径赋值：构造函数或属性。

前面的例子没有给 key_name 属性赋值，所以数据存储为其创建了一个唯一的数字 ID。可以这样查询键：

```
my_entity_key = task.key()
```

输出是类型名后面跟着一个数字值，这里类型名是 Task。此外也可以为实体先准备好一个键，然后在创建时传入。比如说想用 task1 作为 Task 类中某个实体的键，就可以像这样初始化任务实体：

```
another_task = Task(key_name = "task1",
         name = "Yet another task",
         description = "Yet another task is, as the name says, yet another task.",
         start_date = datetime.datetime(2011, 2, 1, 12, 0, 0).date())
```

现在用 another_task.key()查询会返回 Task: task1，里面包含了刚才在创建时赋给 key_name 的值。

在创建 another_task 的例子里，start_date 赋值为 2011/02/01。这是任意选择的一个值，目的是为了说明它可以接受任何有效的日期值。Python 标准模块 datetime.datetime 用来创建正确格式的日期值。datetime.datetime 模块默认使用 UTC 时区创建和读取日期。可以利用模块的能力来设置时区和其他属性。这完全是标准的 Python，可以用你习惯的 Python 方式来操作日期。

下面重新回到模型类，解释包含在代码样例里的一些特性。此外我还会修改模型类来演示其他一些能力。

10.1.2 使用Python进行基本的GAE数据建模

虽然已经给出了第一个基本的模型类，不过更正式和详细的解释还是有些用处。在解释细节时，还会在已经覆盖的内容上继续构建。再看看 Task 模型类：

```
class Task(db.Model):
  name = db.StringProperty(required=True)
  description = db.StringProperty()
  start_date = db.DateProperty(required=True)
  due_date = db.DateProperty()
  end_date = db.DateProperty()
  tags = db.StringListProperty()
```

taskmanager GAE project

首先注意 Task 模型类继承了 db.Model 类。Model 类（google.appengine.ext.db 模块中）是数据建模 API 提供的三个模型类之一，另外两个分别是 Expando 和 PloyModel。Model 类是三个模型类里最为严格和正式的一个。Model 定义了结构化的数据模型，包含一套明确定义的属性，每个属性的数据类型在设计时就定义好。从某些方面看，定义 Model 类或继承自它很像定义传统的数据库结构。

Task（是一个 Model 类）定义了六个属性，每个都有明确的类型，类型用 Property 类的子类来定义。Python 包装器（SDK 和 API）定义并支持一组属性的数据类型。对应有一组类用来帮助定义数据模型中的属性。每个 Property 类为属性值定义一种数据类型，同时它还定义了如何验证和保存数据。例如，StringProperty 类代表所有 Python 中最多不超过 500 个字符的 str 或 unicode 值类型。DataProperty 作为 DateTimeProperty 的子类，只代表日期和时间值中的日期部分。StringListProperty 代表字符串值的列表。

10

GAE Python API 的在线文档中有一节里包括所有支持的值类型。这一节的标题是“属性和值类型”（Properties and Values）。在线文档请访问 http://code.google.com/appengine/docs/python/datastore/entities.html#Properties_and_Value_Types。访问 http://code.google.com/appengine/docs/python/datastore/typesandpropertyclasses.html 可获取对应的类型和属性类的列表。表 10-2 总结了最常见的支持类型和对应的类。

表10-2 GAE Python API中的属性类型和对应的类

值 类 型	属 性 类	排 序	备 注	GAE API定义数据类型
str, unicode	StringProperty	Unicode	小于500个字符，str按ASCII处理	否
db.Text	TextProperty	不可排序	长字符串（长于500字符）	是
db.ByteString	ByteStringProperty	字节序	最大500字节，Db.ByteString 扩展str，表示未编码的byte串	是

（续）

值　类　型	属　性　类	排　　序	备　　注	GAE API定义数据类型
db.Blob	BlobProperty	不可排序	最大1MB	是
Bool	BooleanProperty	False < True		否
int, long(64 bit)	IntegerProperty	数字顺序		否
Float	FloatProperty	数字顺序	float和int同时存在时，int < float，意味着5 < 4.5	否
datetime.datetime	DateTimeProperty DateProperty TimeProperty	时间顺序		否
支持类型的list	ListProperty StringListProperty	升序	升序按最小元素排；降序按最大元素排	否
Null			Python里的None	否

> "No"表明对应的数据类型在 GAE Python API 里没有定义，但是在 Python 语言及其标准库里有定义。

除了表 10-2 中列出的常见数据类型外，GAE Python API 还支持一些类型，用来定义实体键，表示 Google 账户及典型的通信标识符，包括电子邮件地址、即时通信、邮政地址和电话号码。GAE Python API 还定义了表示地理位置、标签或评分值的类。数据存储里的键用 `google.appengine.ext.db` 模块中的 `Key` 类表示。其他支持类型如下。

- Google 账号：`users.User`
- 电子邮件：`db.Email`
- 即时通信：`db.IM`（即时通信 ID）
- 邮政地址：`db.PostalAddress`
- 电话号码：`db.PhoneNumber`
- 分类：`db.Category`
- 链接：`db.Link`
- 评分：`db.Rating`
- 地理位置：`db.GeoPt`

尽管有了 `Model` 和其他支持类型的帮助，可以准确地定义需要的数据架构，但有时灵活性对模型也很重要。你可能还记得，底层数据存储没有任何架构或者数据类型的形式限制。换句话说，允许按需添加属性，同类两个实体的属性集可以有所变化。此外，两个实体也可以选择在同一属性中保存不同数据类型。为了表达如此动态和灵活的架构，GAE Python API 定义了模

型类 Expando。

> Google App Engine 还提供了一个 Blobstore。和数据存储不同，Blobstore 服务支持存储对数据存储而言太大的对象。Blobstore 里的大对象使用 blobstore.BlobKey 来标识。BlobKey 可以按字节排序。

1. Expando

属性可以是下面两种：

- 固定属性
- 动态属性

定义在模型类里的属性是固定属性。添加到模型实例上的属性算是动态属性。

> 一个模型实例，而非类，持久化为一个实例。

继承自 Expando 的模型类的实例可以同时拥有固定的和动态的属性。这使得两个实例（保存为实体）的同一属性可以有不同的数据类型。它也使同一个属性（例如 new_attribute），一个实例添加而另一个实例不添加成为可能。实例可以包含一个新属性但不设置它。我重构了 Task 模型类使它继承自 Expando。新 Task 及其实例的代码如下：

```
import datetime
from google.appengine.ext import db

class Task(db.Expando):
  name = db.StringProperty(required=True)
  description = db.StringProperty()
  start_date = db.DateProperty(required=True)
  due_date = db.DateProperty()
  end_date = db.DateProperty()
  tags = db.StringListProperty()

t1 = Task(name="task1", start_date=datetime.datetime.now().date())
t1.description = "this is task 1"
t1.tags = ["important", "sample"]
t1.collaborator = "John Doe"

t2 = Task(name="task2", start_date=datetime.datetime.now().date())
t2.description = "this is task 2"
t2.tags = ["important", "sample"]
t2.resources = ["resource1", "resource2"]
```

taskmanager GAE project

这个例子不言自明，展示了灵活的 Expando 模型的力量。不过灵活性是有代价的。动态属性不会像固定属性那样得到验证。建模 API 提供了另外一个模型类的变种，来支持定义多态行为。

2. PolyModel

PloyModel 类（在 google.appengine.ext.db.polymodel 模块中）支持定义模型类之间的继承层次结构。通过类继承建立好了层次结构，就可以查询一个类，并在结果集里得到符合条件的类及其子类。为了说明这一点，再次修改 Task 类。我们重构 Task 类继承自 PolyModel 类，然后再创建两个 Task 类的子类。子类 IndividualTask 和 TeamTask 分别代表个人所有者的任务和团队的任务。样例代码如下：

```
from google.appengine.ext import db
from google.appengine.ext.db import polymodel

class Task(polymodel.PolyModel):
  name = db.StringProperty(required=True)
  description = db.StringProperty()
  start_date = db.DateProperty(required=True)
  due_date = db.DateProperty()
  end_date = db.DateProperty()
  tags = db.StringListProperty()

class IndividualTask(Task):
  owner = db.StringProperty()

class TeamTask(Task):
  team_name = db.StringProperty()
  collaborators = db.StringListProperty()
```

taskmanager GAE project

现在如果查询 Task 实体，结果集里除了 Task 实体外，还会包括 IndividualTask 和 TeamTask 实体。理解 App Engine 的查询机制会帮助你更好地理解这个行为。下面我们来介绍查询和索引。

10.1.3　查询与索引

App Engine 提供了类 SQL 的查询语言，称作 GQL。尽管并不完全具备 SQL 的能力，但 GQL 的语法语义还是非常贴近我们习惯使用的 SQL 的。GQL 可以用来查询实体及其属性。实体表现为 GAE Python 和 Java SDK 里的对象。因此 GQL 非常类似于面对对象查询语言，能用来查询、过滤和获取模型实例及其属性。Java 持久化查询语言（Java Persistence Query Language，JPQL，参见 http://download.oracle.com/javaee/5/tutorial/doc/bnbtg.html）是流行的面对对象查询语言的一个例子。

要获取五个 start_time 为 2011 年 1 月 1 日的 Task 实体，并列出它们的名字，可以查询如下：

```
q = db.GqlQuery("SELECT * FROM Task" +
                 "WHERE start_date = :1", datetime.datetime(2011, 1, 1, 12, 0,
0).date())
results = q.fetch(5)
for task in results:
  print "Task name: %s" % (task.name)
```

taskmanager GAE project

或者，可以使用 Query 接口查询到同样的结果：

```
q = Task.all()
q.filter("start_date =", datetime.datetime(2011, 1, 1, 12, 0, 0).date())
results = q.fetch(5)
for task in results:
  print "Task name: %s" % (task.name)
```

taskmanager GAE project

第一个选择是使用 GqlQuery 接口，第二个是使用 Query 接口。两种情况中，都声明了过滤条件来缩小结果集，只包含那些 start_date 属性匹配指定日期的实体。这和在 SQL 的 where 子句里添加条件类似。前面的例子里，2011 年 1 月 1 日中午 12 点被用来作为参数。时间部分可以是任何其他值，比如上午 10 点或者晚上 8 点，参数有效的部分都一样，用到的只有参数的日期部分。

App Engine 支持非常丰富的过滤条件，这些会在下一节里介绍。

前面例子的结果是用 fetch 方法获取的。方法 fetch 接受 limit 参数来限制结果集。此外，fetch 方法还支持可选参数 offset。因此，在例子里可以用 fetch(limit=5, offset=10)代替 fetch(5)返回第 11 到第 15 条记录，而非头 5 条记录。这自然把我们带到了顺序的概念上，一个显而易见的问题就是："结果集里实体的顺序是什么？"因为没有显式声明任何排序条件，所以结果集的顺序不是确定的，而且在不同查询间会发生变化。为了确保特定的顺序，可以在查询里添加排序。例如可以按 name 对结果集排序如下：

```
q = db.GqlQuery("SELECT * FROM Task" +
               "WHERE start_date = :1" +
               "ORDER BY name", datetime.datetime(2011, 1, 1, 12, 0, 0).date())
```

taskmanager GAE project

你可能还记得 Bigtable 存储行记录是有序的。因此，查找特定行记录并不涉及随机读。相反，行键可以被用来定位保存行数据（或实体）的 region sever，行数据可以被顺序读取。使用实体的一个属性来过滤集合时，会查找对应的索引，索引中记录按照期望的顺序被保存成行。如果一个查询按 start_date 属性过滤，按 name 属性排序访问 Task 实体，它就会用到这样一个索引，其中数据按预先排好的顺序保存，先按 start_date 再按 name。事实上，每个有效的查询都是在底层索引的帮助下完成的。换句话说，如果没有对应的索引，任何查询都不能执行。有些查询看起来不同，但是会使用同一个索引。App Engine 会创建一些隐式的索引，特别是那些会涉及用

相等操作符对属性值、键或祖先进行过滤的字段。对那些按多个字段过滤，或者涉及不相等比较，或者要按多个属性进行排序的查询，需要明确地定义索引。开发服务器会帮助你找出需要的索引，并在执行相应查询时创建索引。索引显式定义在 index.yaml 配置文件中。

下面解释所支持的过滤操作。

10.1.4 过滤和结果排序

应用服务引擎支持在属性值上使用下列操作符：

- =
- >
- >=
- <
- <=
- !
- IN

为了匹配一个不等式过滤器，索引被扫描以找出第一个匹配行，从那开始连续返回行记录，直到有一行不匹配为止。在索引里，所有行都是按过滤属性排序的。可以在单个属性上使用多个不定式，但是不能在一个查询中使用多个属性定义多个不等式。单个属性上的多个不等式会被分成多个查询，结果集会在返回前被合并起来。因此下面这个查询：

```
SELECT * FROM Task WHERE start_date >= :a_specified_date
                    AND start_date <= :another_specified_date
```

会分为两部分执行，其中一部分匹配所有 start_date 大于等于 a_specified_date 的行，而另一部分匹配所有 start_date 小于等于 another_specified_date 的行。最后，两个查询的结果会合并到一起。

对包含不等式的查询排序时，必须首先按不等式中的属性排序。还可以添加其他排序用的属性，但都要在不等式使用的属性之后。

同一个查询里，在使用一个不等式过滤的同时，还可以在多个属性上使用多个等式过滤。定义 ORDER BY 条件时，首先还是要按不等式使用的属性排序。

GAE 支持属性包含一组值，它还允许两个实体的同一属性包含不同的数据类型。IN 操作符用在列表属性上，测试是否为成员。只要实体的列表中有一个元素满足过滤条件，它就会被返回。例如，a_prop = [1, 2]同时匹配 a_prop =1 和 a_prop = 2。但是如果查询声明条件是 a_prop > 1 和 a_prop < 2，a_prop = [1, 2]就不会匹配，因为尽管每个条件都有一个元素匹配，但是没有元素同时匹配两个条件。多值属性（也就是那些包含列表值的属性）列表中的每个值都会被添加到索引中。这一点，除了前面提到的匹配行外，在排序时还会引入其他一些副作用。多值属性在按升序排列时会使用列表中的最小值，按降序排列时会采用最大值。因此包含[1, 3, 5, 7]的多值属性按升序排列时会被当作 1，而降序排列时会被当作 7。而且[1, 3, 5, 7]同时小于和大于[2, 3, 4, 5, 6]。

两个实体的同一属性可以包含不同数据类型，有些实体甚至可以不包含属性。在按某一属性查询过滤时，没有定义此属性的实体会被跳过。如果希望这样的实体也包含在结果中，至少应该设置实体的属性值为 null（或 Python 里的 None）。查询只会匹配那些与查询过滤器的数据类型相同的实体，即匹配字符串的查询只会查找那些属性是字符串的实体。在单个属性中使用混合数据类型在排序时也会带来一些副作用。数据类型之间是有次序的，比如整数在浮点数前面。因此，如果按某个包含整数和浮点数的属性排序，结果可能会很诡异，因为会出现 5 < 4.5 这样的结果。

早先我提到过每个查询都需要索引。下面是一个查询及其显式定义的索引：

```
q = db.GqlQuery("SELECT * FROM Task" +
                "WHERE start_date >= :1" +
                "tags IN :2" +
                "ORDER BY start_date",
datetime.datetime(2011, 1, 1, 12, 0, 0).date(), ["Important", "Sample"])
```

taskmanager GAE project

```
indexes:
- kind: Task
  properties:
  - name: start_date
  - name: tags
```

index.yaml in taskmanager GAE project

到目前为止，查询结果都是通过 fetch 方法来获取的。fetch 方法支持每次获取一个集合，返回结果集的条目数由参数来定义。如果只想获取一个实体，可以用 get 方法获取按顺序排的第一个实体。如果只需要知道结果的总数，可以用 count 方法。除非超时，否则 count 方法会返回结果中所有实体的总数。App Engine 更适用于能快速返回的响应，因为这样易于扩展。任何超过 30 秒的响应都会超时。

如果需要遍历整个结果集，就需要把 Query 接口当作可迭代的对象。下面是一个例子：

```
q = db.GqlQuery("SELECT * FROM Task" +
                "WHERE start_date = :1" +
                "ORDER BY name", datetime.datetime(2011, 1, 1, 12, 0, 0).date())
for task in q:
  print "Task name: %s" % (task.name)
```

taskmanager GAE project

可迭代对象支持分小块访问结果集，直至接受完所有结果。虽然可迭代对象支持遍历整个结果集，但是它不支持回溯，也不支持从上次获取的位置再继续获取数据。对于这样的增量获取，游标就派上用场了。

获取后可以用查询对象的 cursor 方法得到游标。游标是 base64 编码的数据指针，支持增量获取数据结果。只要过滤条件、排序规则和祖先一样，两次增量获取结果的查询应该是一样的。执行查询前，要先把游标传递给查询对象的 with_cursor 方法。已经获取过数据的任何修改都

不会影响游标，游标之前范围里的所有更新或插入会都也被忽略。

为了方便保持数据状态一致，App Engine 支持实体和实体组级别的事务。事务完整性意味着操作要么成功要么回滚。如果是单个实体，所有写（即创建、更新和删除操作）都是原子的。

一个实体和它的祖先以及所有子实体形成一个实体组。操作整个组内实体的函数可以包含在一个事务里。只要把函数作为参数传给 db.run_in_transaction 方法，它就可以显式地作为一个事务单元执行。代码清单 10-1 展示了一个例子。

代码清单 10-1　任务管理器 GAE 项目

```
import datetime
from google.appengine.ext import db

class Task(db.Model):
  name = db.StringProperty(required=True)
  description = db.StringProperty()
  start_date = db.DateProperty(required=True)
  due_date = db.DateProperty()
  end_date = db.DateProperty()
  tags = db.StringListProperty()
  status = db.StringProperty(choices=('in progress', 'complete', 'not started'))

def update_as_complete(key, status):
  obj = db.get(key)
  if status == 'complete':
    obj.status = 'complete'
    obj.end_date = datetime.datetime.now().day()

  obj.put()

q = db.GqlQuery("SELECT * FROM Task" +
                "WHERE name = :1", "task1")
completed_task = q.get()

db.run_in_transaction(update_as_complete, completed_task.key(), "complete")
```

jtaskmanager GAE project

App Engine 并不对行记录加锁，而是基于最后一次更新的时间通过乐观锁和协调来解决冲突。它不支持在单个事务内操作多个根实体。

浏览过 App Engine 及其 Python SDK 的大部分基本特性后，现在让我们来了解一下 Java SDK 的特性。

10.1.5　Java App Engine SDK

上手前，请先阅读在线入门教程，地址是 http://code.google.com/appengine/docs/java/gettingstarted/。用 Java 编写的运行在 App Engine 上的 Web 应用用到了 Java 标准，例如 Java Servlet。App Engine 运行时上运行有 Java 应用程序服务器。容器本身是 Webtide Jetty 应用程序服务器的定制版。

无论用 Python 还是 Java 访问，App Engine 的基础都不会变，所以再重复前面的内容没什么意思。所以这一节直接展示如何使用 Java 标准（比如 JDO 和 JPA）来访问数据存储。

App Engine 开源插件 DataNucleus（www.datanucleus.org/）填补了 Java 标准持久化框架（尤其是 JDO 和 JPA）和基于 Google Bigtable 的数据存储之间的空白。

JDO 的设置及配置可以参照在线文档 http://code.google.com/appengine/docs/java/datastore/jdo/。JPA 配置参见 http://code.google.com/appengine/docs/java/datastore/jpa/。

Python 例子里的 `Task` 类可以改成一个 JDO 感知的简单 Java 对象：

```
package taskmanager;

import com.google.appengine.api.datastore.Key;
import java.util.Date;
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;

@PersistenceCapable
public class Task {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    private String name;

    @Persistent
    private String description;

    @Persistent
    private Date startDate;

    @Persistent
    private String status;

    public Greeting(String name, String description, Date startDate,
String status) {
        this.name = name;
        this.description = description;
        this.startDate = startDate;
        this.status = status;
    }

    public Key getKey() {
        return key;
    }

    public User getName() {
```

10

```
        return name;
    }

    public String getDescription() {
        return description;
    }

    public Date getStartDate() {
        return startDate;
    }

    public String getStatus() {
        return status;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public void setStartDate(Date startDate) {
        this.startDate = startDate;
    }

    public void setStatus(String status) {
        this.status = status;
    }
}
```

jtaskmanager GAE project

PersistenceManager 类用于处理实体的持久化。需要通过 PersistenceManagerFactory 获取 PersistenceManager 实例，像这样：

```
package taskmanager;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManagerFactory;

public final class PMF {
    private static final PersistenceManagerFactory pmfInstance =

        JDOHelper.getPersistenceManagerFactory("transactions-optional");

    private PMF() {}

    public static PersistenceManagerFactory get() {
        return pmfInstance;
    }
}
```

jtaskmanager GAE project

最后，可以保存对象如下：

```
String name = "task1";
String description = "a task";
Date startDate = new Date();
String status = "task created";

Task task = new Task(name, description, startDate, status);
PersistenceManager pm = PMF.get().getPersistenceManager();
try {
    pm.makePersistent(task);
    } finally {
    pm.close();
    }
```

jtaskmanager GAE project

接下来可以用类似 GQL 的 JDO 查询语言（JDO Query Language，JDOQL）查询所有任务：

```
PersistenceManager pm = PMF.get().getPersistenceManager();
String query = "select from " + Task.class.getName();
List<Task> tasks = (List<Task>) pm.newQuery(query).execute();
```

jtaskmanager GAE project

使用 JDO 和 JPA（没有在本章中说明）填补了典型的对象为中心的应用程序开发和可扩展的有序列族存储（例如 GAE 的数据存储）之间的空白。它能帮助开发者利用 App Engine 的可扩展环境，而无需学习全新的数据库技术。但是要注意，JDO 和 JPA 标准中只有部分适用于 App Engine。

所有与查询有关的解释（包括这些查询的行为及局限性）对 Python 和 Java SDK 都是一样的。索引、事务及相关概念也是一样的。

下面我们来介绍 Amazon SimpleDB。

10

10.2 Amazon SimpleDB

前一节里我们看到 GAE 数据存储提供了一个完全托管的数据库供开发者使用。管理大型可扩展数据库的复杂性和负担完全被抽象掉了。你无需关心数据库管理、数据库索引管理或性能调优。对于数据存储，你所要做的就是专注在应用程序和数据的逻辑上。

Amazon SimpleDB 是 GAE 数据存储之外另一个即刻可用的数据库。它有弹性，而且完全托管在云中。GAE 数据存储和 SimpleDB 的 API 及内部结构千差万别，但是二者都提供了高可扩展和按需增长的数据存储模型。

> Amazon EC2 database AMI 允许你在 AWS 云上使用自己喜欢的数据库（Oracle、MySQL、PostgreSQL、DB2 或任何其他数据库），但是管理的负担也是你自己的。

10.2.1　SimpleDB入门

Amazon SimpleDB 是 Amazon Web Service（AWS）的一部分。入门非常简单，就是在 http://aws.amazon.com/sdb 上创建一个 SimpleDB 账号。访问 AWS 需要两组凭证：access key 和 secret key。登录自己的 http://aws.amazon.com/page 页面后，可以从账号详情页上得到这些凭证。AWS 注册与访问的细节内容没有包含在本章及本书中，不过按照 AWS 主页（http://aws.amazon.com/）的说明进行应该很容易进行。

SimpleDB 的设计非常简单。它规定了一些限制，同时提供了非常简单的 API 与数据交互。SimpleDB 中最高层次的概念是账户。可以把它看作传统 RDBMS 里的一个数据库实例，或者是包含一系列不同工作簿的 Microsoft Excel 文档。

每个账号可以有一个或多个域，每个域是一个集合。一个 SimpleDB 域（一个集合）默认最多容纳 10GB 数据，每个账号最多可以有 100 个域。这不是最大值，只是默认值。如果需要的话，可以联系 Amazon 要求提供更高容量。默认可以设置 1TB 数据集。这已经不小了！此外，巧妙地组合 SimpleDB 和 Amazon 简单存储服务（Amazon Simple Storage Service，S3）可以帮助优化存储。大对象存储到 S3 里，小对象和大对象的元数据存储到 SimpleDB 里，这样会很管用。

在域里可以持久化条目。条目可以是任意类型，只要它们能定义成属性–值对就行。因此集合中的每个条目都是一个属性–值对。同一个域里的两个条目无需有相同的属性–值对。极端情况下，即使两个条目没有任何共同属性，也可以保存在同一个域里。这种极端例子从 SimpleDB 的角度来看可能没有任何实际用途，但确实是允许的。

前面提到文档数据库也有类似的特点。CouchDB 和 MongoDB 提供了近似的自由度和能力。SimpleDB 可以看作云中的文档数据库，能够按需扩展。在 SimpleDB 里存储下面的日志数据格式（来自第 3 章）很容易，而且很合适：

```
{
    "ApacheLogRecord": {
        "ip": "127.0.0.1",
        "ident" : "-",
        "http_user" : "frank",
        "time" : "10/Oct/2000:13:55:36 -0700",
        "request_line" : {
            "http_method" : "GET",
            "url" : "/apache_pb.gif",
            "http_vers" : "HTTP/1.0",
        },
        "http_response_code" : "200",
        "http_response_size" : "2326",
        "referrer" : "http://www.example.com/start.html",
        "user_agent" : "Mozilla/4.08 [en] (Win98; I ;Nav)",
    },
}
```

这个例子是一个 JSON 文档。文档里的每个键/值对正好对应 SimpleDB 里的一个属性–值对。

> 上面例子的 JSON 格式只是用来演示键/值对。SimpleDB 本身并不能理解 JSON 格式或查询 JSON 文档。JSON 文档必须被解析，并从中提取出键/值对才能存储进 SimpleDB。

和大多数 AWS 功能一样，SimpleDB 提供了简单的 API 来操纵域、条目和条目的属性-值对。API 以 Web 服务方式提供，同时支持 REST 和 SOAP 两种风格。客户端发送请求来执行特定的操作，例如创建域、插入条目或者更新属性-值对。SimpleDB 服务器完成操作，如果不出现错误，就会返回成功信息和应答数据。返回的应答数据是一个 HTTP 响应，包含头部信息、元数据和一些 XML 格式的内容。

下面列出 SimpleDB API 中可用的命令。我们首先来看有助于操作域的命令。

SimpleDB 域管理命令如下。

- ❑ `CreateDomain`：创建一个域来存储条目。
- ❑ `DeleteDomain`：删除已有域。
- ❑ `ListDomain`：列出账号下的所有域。
- ❑ `DomainMetadata`：获取有关域、域的条目及条目的属性–值对的相关信息，包含域创建时间、域的条目总数和属性–值对的大小等。

创建完域以后，可以使用 `PutAttributes` 方法插入或更新条目。条目是属性–值的集合。插入条目意味着创建一组属性–值集合，它们逻辑上形成一个条目。更新一个条目是先获取条目，然后更新其中一个或多个属性。用 `BatchPutAttributes` 能在一次调用里执行多个 `put` 操作。

`DeleteAttributes` 用来删除域的条目、属性–值对或是属性的值。`BatchDeleteAttributes` 支持在一次调用里执行多个 `delete` 操作。

用 `GetAttributes` 操作可以获取一个条目的属性–值对。另外还可以用 `SELECT` 操作来查询、过滤域里的条目。SimpleDB 对域数据支持丰富的过滤操作，其语法语义和 SQL 提供的相似。SimpleDB 会自动创建和管理索引，以便提高查询效率。

虽然 SimpleDB 的查询机制感觉很像 SQL，但是不要混淆 SimpleDB 和 RDBMS。它不是关系型存储，也不像关系型存储那样支持复杂事务，或者基于外键的约束。

10

SimpleDB 区域

目前，AWS 对四个地区提供 SimpleDB：美国东部、美国西部、欧洲和亚洲。在创建域之前要先选择一个地区。选择接近用户的区域能减少延迟、提高性能。不同区域的域名可以一样，但它们其实不同，并且相互完全隔离，不共享任何数据。

这四个区域（及其地理位置）如下。

- ❑ sdb.amazonaws.com：美国东部（北弗吉尼亚州）
- ❑ sdb.us-west-1.amazonaws.com：美国西部（北加州）

- sdb.eu-west-1.amazonaws.com：欧洲（爱尔兰）
- sdb.ap-southeast：亚洲（新加坡）

下面说明访问和使用 SimpleDB 的几种方式。

10.2.2 使用REST API

SimpleDB 最简单的使用方式是 REST API。尽管从一个纯粹主义者的角度看，SimpleDB 的 REST API 并不是彻底 RESTful 的，但它还是提供了简单的基于 HTTP 的请求–响应模型。建议阅读 Subbu Allamaraju 题为“A RESTful version of Amazon SimpleDB”的帖子（www.subbu.org/weblogs/main/2007/12/a*restful*versi.html），了解为什么 SimpleDB REST API 不算真正 RESTful。测试这个 API 最简单的办法是用命令行客户端执行操作。这一节里使用基于 Perl 的命令行客户端，名叫 amazon-simpledb-cli，可以从项目主页下载这个客户端，地址为 http://code.google.com/p/amazon-simpledb-cli/。amazon-simpledb-cli 依赖于 AWS 的 Perl 模块。Perl 模块可以在这里下载：http://aws.amazon.com/code/1136。

> Amazon SimpleDB 还提供了 SOAP API。本书不会介绍 SOAP API，读者可以访问在线开发者文档学习 SOAP API，地址为 http://aws.amazon.com/documentation/simpledb/。

要安装 amazon-simpledb-cli，先确认安装了 Perl。如果你是 POSIX 系统用户（包括各种 Linux、BSD 和 Mac OSX），那可能 Perl 已经装好了。如果没有，需要获取 Perl 编译器和解释器。Perl 的安装说明超出了本书的范围，如果需要帮助，可以访问 perl.org。

开始之前，确认已获取下列 Perl 模块：

- `Getopt::Long`
- `Pod::Usage`
- `Digest::SHA1`
- `Digest::HMAC`
- `XML::Simple`
- `Bundle::LWP`
- `Crypt::SSLeay`

像这样安装 `Getopt::Long`：

```
perl -MCPAN -e 'install Getopt:Long'
```

用同样的方法安装其他需要的 Perl 模块。记得用其他模块的名字替换 `Getpot::Long`。在某些系统上，某些模块可能需要以管理员账号安装。安装和更新完必需模块后，可以像下面这样安装 AWS Perl 模块。

(1) 解压下载文件：`unzip AmazonSimpleDB-*-perllibrary.zip`。

(2) 获取 Perl sitelib：`sitelib=$(perl -MConfig -le 'print $Config{sitelib}')`。

(3) 复制 Amazon 模块到 `sitelib`：

```
sudo scp -r AmazonSimpleDB-*-perl-library/src/Amazon $sitelib
```

安装完 AWS Perl 模块后，获取 `amazon-simpledb-cli` 脚本如下：

```
sudo curl -Lo /usr/local/bin/simpledb http://simpledb-cli.notlong.com
```

设置脚本权限，允许所有人执行脚本：

```
sudo chmod +x /usr/local/bin/simpledb
```

现在就准备好了。接下来，先确认找到了 AWS 凭证（AWS access key 和 AWS access secret key，可以从你的账号页面上找到），以便测试 `amazon-simpledb-cli` 脚本（安装成 `/usr/local/bin` 下面的 `simpledb`）。

要使用 `simpledb` 脚本，需要把 access key 和 secret access key 分别作为 `aws-access-key-id` 和 `aws-secret-access-key` 命令行参数传入。此外还可以用 `$AWS_ACCESS_KEY_ID` 和 `$AWS_SECRET_ACCESS_KEY` 环境变量来设置默认的 access key 和 secret access key。

像这样创建一个域：

```
simpledb create-domain domain1
```

添加条目到域里：

```
simpledb put domain1 item1 key1=valueA key2=value2 anotherKey=someValue
simpledb put domain1 item2 key1=valueB key2=value2 differentKey=aValue
```

编辑 `item1` 并向其中添加另一个属性–值对：

```
simpledb put domain1 item1 yetAnotherKey=anotherValue
```

替换属性–值对：

```
simpledb put-replace domain1 item1 key1=value1 newKey1=newValue1
```

删除属性或属性值，例如：

```
simpledb delete mydomain item1 anotherKey
simpledb delete mydomain item2 key2=value2
```

罗列所有域：

```
simpledb list-domains
```

列出域里的所有条目名：

```
simpledb select 'select itemName() from domain1'
```

或者用类 SQL 语法过滤条目，列出所有匹配的条目及其属性：

```
simpledb select 'select * from domain1 where key1="valueA"'
```

要列出一个特定条目（比方说 `item1`）的所有属性，可以像这样使用 `simpledb`：

```
simpledb get domain1 item1
```

如果要限制输出特定一组属性，可以在上一个命令中传入属性名：

```
simpledb get mydomain item1 newKey1 key2
```

10

如果想要删除一个域及其所有组成部分，可以运行 simpledb 命令如下：

```
simpledb delete-domain domain1
```

认证请求

对 SimpleDB 的每个请求都需要认证。客户端请求要包括下列内容：

- AWS access key
- 基于 AWS secret access key 和请求生成的 HMAC-SHA1 签名
- 时间戳

AWS 根据传入的 AWS access key 访问对应的 secret access key，然后用 secret access key 和传入的请求生成一个 HMAC-SHA1 签名。只有客户端传入的签名和服务器生成的签名匹配，才会返回适当的响应，否则会抛出认证错误。

传入的时间戳作为额外一层安全防护。时间戳超过 15 分钟的请求被认为是过时而不会被处理。

前面的命令让我们体会到可以用 amazon-simpledb-cli 做什么，还介绍了 Amazon SimpleDB 中可用的数据查询和管理命令。

为了完整起见，下面会稍微讲解一下在使用 REST API 时的基本请求和响应。像下面这样的调用：

```
simpledb put domain1 item1 key1=valueA key2=value2 anotherKey=someValue
```

会被翻译成：

```
https://sdb.amazonaws.com/
?Action=PutAttributes
&DomainName=domain1
&ItemName=item1
&Attribute.1.Name=key1
&Attribute.1.Value=valueA
&Attribute.2.Name=key2
&Attribute.2.Value=value2
&Attribute.3.Name=anotherKey
&Attribute.3.Value=someValue
&AWSAccessKeyId=[valid access key id]
&SignatureVersion=2
&SignatureMethod=HmacSHA256
&Timestamp=2011-01-29T15%3A03%3A05-07%3A0C
&Version=2009-04-15
&Signature=[valid signature]
```

它的响应是 XML 文档，格式如下：

```
<PutAttributesResponse>
  <ResponseMetadata>
    <RequestId></RequestId>
    <BoxUsage></BoxUsage>
  </ResponseMetadata>
</PutAttributesResponse>
```

Amazon SimpleDB XSD 可以在这里找到：http://sdb.amazonaws.com/doc/2009-04-15/AmazonSimpleDB.xsd，响应 XML 的结构定义在这个文档里。

以上我们介绍了 SimpleDB 的不少特性，接下来介绍用 Java、Python 和 Ruby 类库访问 SimpleDB。

10.2.3 使用Java访问SimpleDB

AWS 为 Java 开发者们提供了全面的 SDK 来编写应用程序与 AWS 交互，而且支持做得也不错。AWS SDK for Java 可以在线获取，地址：http://aws.amazon.com/sdkforjava/。开始可以先阅读 SDK 入门文档，地址：http://aws.amazon.com/articles/3586。这个 SDK 支持很多 AWS 服务，包括 SimpleDB。下载文件中还包含一些入门样例。代码清单 10-2 是一个展现 SDK 之应用的基本例子。

代码清单 10-2 使用 AWS SDK 与 SimpleDB 交互的简单 Java 程序

```
import java.util.ArrayList;
import java.util.List;
import com.amazonaws.AmazonClientException;
import com.amazonaws.AmazonServiceException;
import com.amazonaws.auth.PropertiesCredentials;
import com.amazonaws.services.simpledb.AmazonSimpleDB;
import com.amazonaws.services.simpledb.AmazonSimpleDBClient;
import com.amazonaws.services.simpledb.model.Attribute;
import com.amazonaws.services.simpledb.model.BatchPutAttributesRequest;
import com.amazonaws.services.simpledb.model.CreateDomainRequest;
import com.amazonaws.services.simpledb.model.Item;
import com.amazonaws.services.simpledb.model.ReplaceableAttribute;
import com.amazonaws.services.simpledb.model.ReplaceableItem;

public class SimpleDBExample {

    public static void main(String[] args) throws Exception {
        AmazonSimpleDB sdb = new AmazonSimpleDBClient(new PropertiesCredentials(
SimpleDBExample.class.getResourceAsStream("aws_credentials.properties")));

        try {
            String aDomain = "domain1";
            sdb.createDomain(new CreateDomainRequest(aDomain));

            // Put data into a domain
            sdb.batchPutAttributes(new BatchPutAttributesRequest(myDomain,
createSampleData()));
        } catch (AmazonServiceException ase) {
            System.out.println("Error Message:    " + ase.getMessage());
            System.out.println("HTTP Status Code: " + ase.getStatusCode());
            System.out.println("AWS Error Code:   " + ase.getErrorCode());
            System.out.println("Error Type:       " + ase.getErrorType());
            System.out.println("Request ID:       " + ase.getRequestId());
        } catch (AmazonClientException ace) {
            System.out.println("Error Message: " + ace.getMessage());
        }
```

```
    }

    private static List<ReplaceableItem> createSampleData() {
        List<ReplaceableItem> myData = new ArrayList<ReplaceableItem>();

        sampleData.add(new ReplaceableItem("item1").withAttributes(
                new ReplaceableAttribute("key1", "valueA", true),
                new ReplaceableAttribute("key2", "value2", true),
                new ReplaceableAttribute("anotherKey", "someValue", true)
                );

        sampleData.add(new ReplaceableItem("item2").withAttributes(
                new ReplaceableAttribute("key1", "valueB", true),
                new ReplaceableAttribute("key2", "value2", true),
                new ReplaceableAttribute("differentKey", "aValue", true)
                );
        return myData;
    }
}
```

SimpleDBExample.java

代码清单 10-2 的例子假设在名为 `aws_credentials.properties` 的文件里保存了 AWS 凭证。`aws_credentials.properties` 文件内容如下：

```
accessKey =
secretKey =
```

这个例子用 Java 程序来演示 API 的用法。如果程序更复杂，就可能会用上标准 Java 组件，包括 Java 持久化 API（JPA）。可以借助一些开源项目来使用 JPA 持久化数据到 SimpleDB。SimpleJPA 就是这样一个项目，它涵盖了与 SimpleDB 有关的 JPA 子集。

10.2.4　通过Ruby和Python使用SimpleDB

Rails 是 Ruby 社区的 Web 开发之选。如果想在 Rails 应用里使用 SimpleDB，没有外界的帮助，肯定没法将原来的关系型数据库（例如 MySQL）简单替换成 SimpleDB。不过 SimpleRecord 可以解决这个问题。可以访问 SimpleRecord 同名开源项目来获取源代码，地址：https://github.com/appoxy/simple_record/。想用 Amazon SimpleDB 作为持久化存储的 Rails 应用，可以用 SimpleRecord 替代 ActiveRecord。

使用 SimpleRecord 很容易，其安装也就是一行代码的事儿：

```
gem install simple_record
```

假定已经安装了 Ruby、RubyGems 和 Rails。最简单的例子如下：

```
require 'simple_record'
 class MyModel < SimpleRecord::Base
    has_strings :key1
    has_ints :key2
 end
```

和前面一样，要先配置 AWS 凭证以准备好使用 SimpleDB。配置 AWS 凭证如下：

```
AWS_ACCESS_KEY_ID='<aws_access_key_id>'
 AWS_SECRET_ACCESS_KEY='<aws_secret_access_key>'
 SimpleRecord.establish_connection(AWS_ACCESS_KEY_ID,AWS_SECRET_ACCESS_KEY)
```

可以存储实例了，样例如下：

```
m_instance = MyModel.new
 m_instance.key1 = "valueA"
 m_instance.key2 = value1
 m_instance.save
```

可以按标识符获取实例：

```
m_instance_2 = MyModel.find(id)
```

此外，还可以找出匹配条件的所有实例，像这样：

```
all_instances = MyModel?.find(:all, ["key1=?", "valueA"],
:order=>"key2", :limit=>10)
```

上面介绍了如何在 Rails 应用中使用 SimpleDB。此外还有一些替代的类库，包括 Amazon 提供的用来连接 Amazon 服务的 Ruby 语言接口。可以从网上下载这个 SimpleDB 的 Ruby 类库来进一步了解，地址：http://aws.amazon.com/code/Amazon-SimpleDB/3324。

最后我们来介绍如何用 Python 访问 AWS SimpleDB。Boto 是访问 SimpleDB 的 Pyhon 类库中最流行的选择，可以从 http://code.google.com/p/boto/获取。开始前先从 Github 镜像下载最新的 boto 源代码：

```
git clone https://github.com/boto/boto.git
```

然后在克隆的版本库目录下运行 `python install setup.py` 来安装 boto。装好后启动 Python 命令行，可以很容易地创建一个新域，然后往里面添加条目，像这样：

```
import boto
 sdb = boto.connect_sdb('<your aws access key>', '<your aws secret key'>)
 domain = sdb.create_domain('domain2')
 item = domain.new_item('item1')
 item['key1'] = 'value1'
 item['key2'] = 'value2'
 item.save()
```

除此以外，SimpleDB 命令和交互的方式与前面见过的其他例子都保持一致。

10.3 小结

本章涵盖了两个流行的、可伸缩的数据库云服务，说明了它们的行为特点和特殊之处，还介绍了怎样结合各种语言的类库、指南和框架使用 NoSQL 存储。

Google App Engine 数据存储和 Amazon SimpleDB 对数据库领域而言都是革命性的，它们促使每个人重新思考他们为管理数据库所作出的各种努力。对许多人来说，站在巨人的肩膀上实现可扩展架构不仅非常诱人，而且从成本和灵活性的角度看也是实用和审慎的。

10

Google 和 Amazon 提供的产品是它们这一类中最广为人知的和健壮的。更多云数据库的选择正在开始涌现。例如，CouchDB 和 MongoDB 的弹性云数据库主机已经诞生了。CouchDB 的创造者推出的主机叫做 CouchOne（www.couchone.com）。与之相似，MongoHQ 是可伸缩的 MongoDB 宿主。文档数据库也不是唯一的可扩展托管选择。最终一致性键/值数据库的创建者 Basho 和 Joyent 一起提供了 Riak 的 3、5 节点集群。希望将来还能见到更多的选择。

随着云计算被越来越多地采纳，我们很可能会看到很多数据库云服务。绝大多数或者至少很多云数据库都会用上 NoSQL 产品。这会为许多开发者提供使用 NoSQL 的机会，并使得开发者开始把数据库看作持久化服务。

第 11 章

MapReduce 可扩展并行处理

本章内容

- 了解可扩展并行处理的挑战
- 利用 MapReduce 进行大规模并行处理
- 探索 MapReduce 计算模型的概念和精妙之处
- 通过 MongoDB、CouchDB 和 HBase 使用 MapReduce
- 介绍基于 MapReduce 的机器学习框架 Mahout

操纵大量数据要求使用的方法和工具能并行执行，而且之间的交互点要尽可能少。更少的交互点意味着更少的潜在冲突和更少的管理。这样的并行处理工具同时还要尽可能降低传输的数据量。I/O 和带宽往往会成为阻碍快速高效处理的瓶颈。I/O 瓶颈在大数据量下会变得更为凸显，并且有可能会拖慢整个系统，以至于系统没法再用。因此对大规模计算而言，保持数据本地计算就显得极其重要。考虑到如此多的方面，处理多台机器上的大数据集就显得既重要又不简单了。

多年来，形形色色的方法被开发出来，用于计算大数据集。最初，创新的重点在于构建超级计算机。超级计算机就是超级给力的计算机，具备远超正常水平的处理能力。这些机器能很好地运行特别设计的复杂的计算密集型算法，不过离足够好的通用解决方案还差得很远。它们的建造和维护成本高昂，对绝大部分组织来说都遥不可及。

网格计算的出现，为解决超级计算机的难题带来了新的曙光。计算网格的想法是把工作分散到一系列节点上进行，减少原来单台机器完成同一工作所需的时间。网格计算的重点转移到了消息传递接口（Message Passing Interface，MPI）上，使用 MPI 或其变种实现数据在节点之间传递，最终完成计算密集型任务。如果提高 CPU 频率能加速完成工作，这种拓扑就能很好地工作。但是如果节点间要传递大量数据，效率就会降低。大数据量传输会受到 I/O 和带宽限制，现实情况也常常如此。此外，管理数据共享逻辑和失败恢复的责任也完全落在了开发者身上。

SETI@Home（http://setiathome.berkeley.edu/）和 Folding@Home（http://folding.stanford.edu/）等公共计算项目对网格计算的想法进行了扩展，让个人贡献出“空闲”的 CPU 周期来帮助处理计算密集型任务。这些项目利用空闲的 CPU 时间来运行，CPU 周期来自志愿者提供的成百上千，有时甚至是上百万台个人机器。这些机器与互联网的连接时开时断，虽然每个个体未必可靠，但是仍然组成了巨大的计算机集群。通过合并空闲的 CPU，整个设施总体上更像是单个超级计算

机，甚至常常超过超级计算机。

虽然有各种不同的有效分布式计算解决方案，但是它们中没有一个能将数据保持在计算网格本地来减小带宽堵塞。很少有人会遵循在节点间不共享或者尽量少共享的策略。MapReduce 的灵感来自于函数式编程的理念，也就是说坚持在并行进程或线程之间尽量减少相互依赖，同时尽力将数据和计算保持在一起。MapReduce 主要用于分布式计算，并由 Google 持有专利。作为一种高效可靠的大数据处理方法，MapReduce 如今已经成为最流行的方法之一。MapReduce 提供了简单、容错的模型，以便有效地计算分布在集群上的大数据，集群可由商用机器横向扩展而成。本章将解释 MapReduce，并探讨这种编程模型在大数据上可以进行的计算。

> MapReduce 的 camel-case 版本由 Google 使用和推广。不过，这里所覆盖的内容更为通用，且不限于 Google 的定义。
>
> MapReduce 的想法发布在一篇研究论文中，论文在线地址：http://labs.google.com/papers/mapreduce.html（Dean, Jeffrey & Ghemawat, Sanjay (2004), “MapReduce: Simplifi ed Data Processing on Large Clusters”）。

11.1 理解 MapReduce

第 6 章曾介绍过用 MapReduce 分组 MongoDB 集群上的数据，因此 MapReduce 对你并不是完全陌生的。不过为了解释 MapReduce 的细节和习惯用法，下面我们用一些例子来重新介绍。

开始先用 MapReduce 运行一些查询，查询涉及聚合函数，比如 `sum`、`maximum`、`minimum` 和 `average`。要用到 1970 年到 2010 年间 NYSE 的每日市场数据，这些数据对外公开。因为数据是按日汇总，每只股票每个交易日只有一个数据点，所以数据集并不大，肯定没大到大数据的程度。样例主要是关注 MapReduce 基础，所以大小不重要。这个例子里使用两种文档数据库：MongoDB 和 CouchDB。MapReduce 不特定于这些产品，而是适用于许多 NoSQL 产品，包括有序列族存储和分布式键/值映射。因为文档数据库的安装配置不用花特别多的精力，而且很容易在本地模式下测试，我们所以先从文档数据库开始。稍后再介绍通过 Hadoop 和 HBase 上使用 MapReduce。

开始前先下载 1970 年至 2010 年的 NYSE 市场数据，地址：http://infochimps.com/datasets/daily-1970-2010-open-close-hi-low-andvolume-nyse-exchange。把文件解压到本地目录，其中包含许多文件，分两类：每日市场数据文件和股息数据文件。为了简单，只加载每日市场数据文件到数据库中。即只需要那些名字以 `NYSE_daily_prices_` 开头，且结尾是一个数字或字母的文件。这些文件中末尾有一个数字的只包含头信息，因而可以跳过。

MongoDB 数据库和集合分别命名为 `mydb` 和 `nyse`，CouchDB 数据库命名为 `nyse`。数据是 CSV 格式的，可以利用 monogoimport 工具将数据导入到 MongoDB 集合中。后面会用 Python 脚本把这个 `.csv` 文件加载到 CouchDB 中。

加载 `NYSE_daily_prices_A.csv` 的 `mongoimport` 命令及其输出如下：

```
~/Applications/mongodb/bin/mongoimport --type csv --db mydb --collection nyse --
headerline NYSE_daily_prices_A.csv
connected to: 127.0.0.1
    4981480/40990992 12%
      89700 29900/second
    10357231/40990992 25%
      185900 30983/second
    15484231/40990992 37%
      278000 30888/second
    20647430/40990992 50%
      370100 30841/second
    25727124/40990992 62%
      462300 30820/second
    30439300/40990992 74%
      546600 30366/second
    35669019/40990992 87%
      639600 30457/second
    40652285/40990992 99%
      729100 30379/second
imported 735027 objects
```

其他文件用同样的方法加载。为了避免顺序上传 36 个不同文件的繁琐，可以考虑用 shell 脚本自动执行任务，脚本如代码清单 11-1 所示。

代码清单 11-1　infochimps_nyse_data_loader.sh

```
#!/bin/bash
FILES=./infochimps_dataset_4778_download_16677/NYSE/NYSE_daily_prices_*.csv
for f in $FILES
do
  echo "Processing $f file..."
  # set MONGODB_HOME environment variable to point to the MongoDB installation
folder.
  ls -l $f
  $MONGODB_HOME/bin/mongoimport --type csv --db mydb --collection nyse --
headerline $f
Done
```

infochimps_nyse_data_loader.sh

加载完数据以后，可以查看单个文档来检查格式：

```
> db.nyse.findOne();
{
       "_id" : ObjectId("4d519529e883c3755b5f7760"),
       "exchange" : "NYSE",
       "stock_symbol" : "FDI",
       "date" : "1997-02-28",
       "stock_price_open" : 11.11,
       "stock_price_high" : 11.11,
       "stock_price_low" : 11.01,
       "stock_price_close" : 11.01,
       "stock_volume" : 4200,
       "stock_price_adj_close" : 4.54
}
```

下面用 MapReduce 来处理集合。第一个任务是找出 1970 年至 2010 年间每股最高价。

MapReduce 包括两部分：map 函数和 reduce 函数。尽管底层系统频繁地以并行方式执行计算，但这两个函数是顺序应用到数据上的。Map 接受一个键/值对然后发出另一个键/值对。Reduce 接受 map 阶段的输出，通过处理这些键/值对产生最后的结果。map 函数会应用到集合的每个条目上，集合可以大到分布在多台物理机器上。map 函数运行在分布式节点本地集合的子集上。一个节点上的 map 操作和另一个节点上的 map 操作完全独立。这种清晰的隔离提供了高效的并行处理，同时还能支持失败时在子集上重新运行 map 函数。

map 函数在整个集合上运行完以后，创建出来的值传递给 reduce 阶段。MapReduce 框架会处理从多个节点收集和整理输出的工作，并使其可以从一个阶段传递到下一个阶段。

Reduce 函数接收 map 阶段生成的键/值对，做进一步处理得到最终结果。Reduce 阶段可能会基于一个公共键来聚合值。Reduce 类似 map，也运行在分布式集群的每个节点上。不同节点上 reduce 操作的结果会合并起来产生最终结果。每个节点执行的 reduce 操作独立于其他节点，当然最后的合并除外。

键/值对可以经历多遍 map 和 reduce 阶段，以支持对已分组和聚合过的数据进行再聚合、再处理。对给定数据集，当需要多种不同的汇总数据时，这种情况常会发生。

11.1.1　找出每股最高价

让我们回到第一个任务，即找出 1970 年至 2010 年间每股最高价，合适的 map 函数如下：

```
var map = function() {
  emit(this.stock_symbol, { stock_price_high: this.stock_price_high });
};
```

manipulate_nyse_market_data.txt

函数会应用到集合的所有文档上。对每个文档，取 `stock_symbol` 作为键，并生成 `stock_symbol` 和 `stock_price_high` 组合的键/值对。过程如图 11-1 所示：

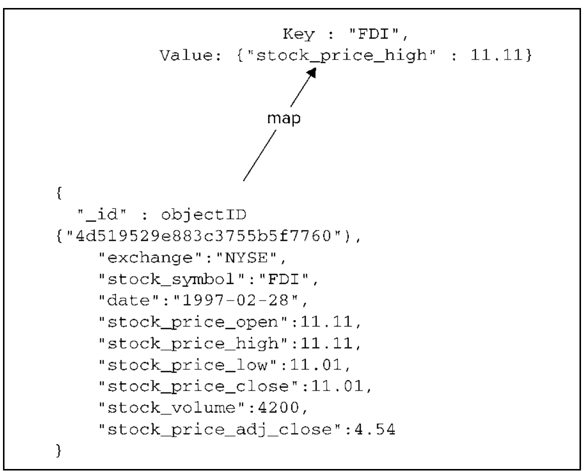


图　11-1

map 阶段提取出的键/值对是 reduce 阶段的输入。在 MongoDB 里，reduce 函数可以定义成下面的 JavaScript 函数：

> MongoDB 只支持用 JavaScript 定义 map 和 reduce 函数。

```
var reduce = function(key, values) {
  var highest_price = 0.0;
  values.forEach(function(doc) {
    if( typeof doc.stock_price_high != "undefined") {
      print("doc.stock_price_high" + doc.stock_price_high);
      if (parseFloat(doc.stock_price_high) > highest_price) { highest_price =
parseFloat(doc.stock_price_high); print("highest_price" + highest_price); }
    }
  });
  return { highest_stock_price: highest_price };
};
```

manipulate_nyse_market_data.txt

reduce 函数接受两个参数：1 个键和 1 个值的数组。在这个例子里，代号"FDI"的股票会在 map 阶段产生许多不同的键/值对。其中一些如下：

```
(key : "FDI", { "stock_price_high" : 11.11 })
(key : "FDI", { "stock_price_high" : 11.18 })
(key : "FDI", { "stock_price_high" : 11.08 })
(key : "FDI", { "stock_price_high" : 10.99 })
(key : "FDI", { "stock_price_high" : 10.89 })
```

算一下总数：`db.nyse.find({stock_symbol: "FDI"}).count();`，发现总共有 5596 条记录。因此 map 阶段应该生成同样数量的键/值对。考虑到有些记录的值没有定义，所以 map 阶段生成的结果可能不是正好 5596 个。

reduce 函数像这样接受值：

```
reduce('FDI', [{stock_price_high: 11.11}, {stock_price_high: 11.18},
 {stock_price_high: 11.08}, {stock_price_high: 10.99}, ...]);
```

此时如果回顾 reduce 函数，可以注意到对每个键都遍历了对应的值数组，并且对数组的每个元素调用了闭包。这个闭包（或者说嵌套函数）执行一个简单的比较，确定出一组值里最高的价格。

reduce 阶段的输出是一组包含股票代号和最高价格的键/值对，每只股票一个键/值对。MongoDB 支持一个可选的 finalize 函数，能传入 reduce 函数的输出并进一步汇总。

下面，在 CouchDB 里加载同样的数据，然后用 MapReduce 执行一些其他类型的聚合函数。

11.1.2 加载历史NYSE市场数据到CouchDB

开始前，需要用脚本解析.csv 文件，将.csv 记录转换成 JSON 文档，然后将其加载入

CouchDB 服务器。用简单的顺序 Python 脚本就可以完成任务，不过顺序加载 900 万文档还是很慢。更实际的情况是用更健壮的并行脚本把数据添加到 CouchDB 里。为了获取最大效率，还可以利用 CouchDB 的批量上传 API 同时上传几千个文档。

脚本的核心功能封装在名为 `upload_nyse_market_data` 的函数里，内容如下：

```
def upload_nyse_market_data():
    couch_server = Couch('localhost', '5984')
    print "\nCreate database 'nyse_db':"
    couch_server.createDb('nyse_db')

    for file in os.listdir(PATH):
        if fnmatch.fnmatch(file, 'NYSE_daily_prices_*.csv'):
            print "opening file: " + file
            f = open(PATH+file, 'r' )
            reader = csv.DictReader( f )
            print "beginning to save json documents converted from csv data in
" + file for row in reader:
                json_doc = json.dumps(row)
                couch_server.saveDoc('nyse_db', json_doc)
                print "available json documents converted from csv data in
" + file + " saved"
                print "closing " + file
            f.close()
```

upload_nyse_market_data_couchdb.py

这个函数解析所有文件名匹配`'NYSE_daily_prices_*.csv'`的文件。Python 脚本利用 `csv.DicReader` 解析 `.csv` 文件并轻松提取头信息，然后用 JSON 模块把解析完的记录转成 JSON 文档。这个函数用 `Couch` 类连接 CouchDB 服务器，创建和删除数据库，添加和删除文档。`Couch` 类是对 CouchDB REST API 的简单封装，从 CouchDB wiki（http://wiki.apache.org/couchdb/Getting *started*with_Python）样例得到了很多启发。

数据加载完以后，用 MapReduce 执行聚合函数。首先重新执行前面 MongoDB 里试过的查询，即找出 1970 年到 2010 年间每股最高价。然后再执行另外一个查询，找出 1970 年至 2010 年之间每只股票每年的最低价格。第一个查询中最高价格要从整个 40 年里找出来，第二个查询则按两个级别聚合数据：年份和股票。

CouchDB 里，对文档数据库进行处理和过滤的 MapReduce 查询会创建视图。视图是 CouchDB 文档查询和报告的主要工具。有两种视图：永久的和临时的。这一节使用永久视图来演示。永久视图会生成底层数据索引，索引建完以后，它的速度就能变快，因此建议在生产环境中使用。临时视图适于原型设计。视图定义在设计文档里。设计文档是特殊类型的 CouchDB 文档，可以执行应用程序代码。CouchDB 支持多视图服务器，即能用不同编程语言编写代码。这意味着可以使用 JavaScript、Erlang、Java 或其他任何所支持的语言为 CouchDB 编写 MapReduce。本节我们用 JavaScript 来说明 CouchDB 的基于 MapReduce 的查询功能。

下面的设计文档包含了三个视图，它们的目的分别如下。

- ❑ 列出所有文档。

- 找出 1970 年至 2010 年之间每股最高价格。
- 找出每只股票每年的最低价格。

设计文档如下：

```
{
  "_id":"_design/marketdata",
  "language": "javascript",
  "views": {
    "all": {
      "map": "function(doc) { emit(null, doc) }"
    },
    "highest_price_per_stock": {
      "map": "function(doc) { emit(doc.stock_symbol, doc.stock_price_high) }",
      "reduce": "function(key, values) {
        highest_price = 0.0;
        for(var i=0; i<values.length; i++) {
          if( (typeof values[i] != 'undefined') && (parseFloat(values[i]) >
highest_price) ) {
                highest_price = parseFloat(values[i]);
          }
        }
        return highest_price;
      }"
    },
    "lowest_price_per_stock_per_year": {
      "map": "function(doc) { emit([doc.stock_symbol, doc.date.substr(0,4)],
doc.stock_price_low) }",
      "reduce": "function(key, values) {
        lowest_price = parseFloat(values[0]);
        for(var i=0; i<values.length; i++) {
          if( (typeof values[i] != 'undefined') && (parseFloat(values[i]) <
lowest_price) ) {
                lowest_price = parseFloat(values[i]);
          }
        }
        return lowest_price;
      }"
    }
  }
}
```

mydesign.json

这个设计文档保存在名为 `mydesign.json` 的文件中，将该文档上传到 `myse_db` 数据库中：

```
curl -X PUT http://127.0.0.1:5984/nyse_db/_design/marketdata -d @mydesign.json
```

CouchDB 的 REST 式交互和 JSON 文档让管理数据库文档变得非常简单，就是编辑、上传设计文档。使用 HTTP PUT 方法上传设计文档会得到下面的响应：

```
{"ok":true,"id":"_design/marketdata","rev":"1-9cce1dac6ab04845dd01802188491459"}
```

响应的部分内容会有所变化，但是如果看见了错误，那说明要么设计文档出了问题，要么上

传操作出了问题。

Futon 是 CouchDB 基于 Web 的管理界面，能用来查看设计文档，调用视图触发 MapReduce 任务。第一次在大量数据上运行 MapReduce 可能比较慢，因为 CouchDB 要基于 map 函数创建索引。后续运行使用索引，执行起来就快多了。Futon 还提供了分阶段视图来展示 map 和 reduce 任务，对理解数据的聚合过程非常有用。

前面例子的聚合逻辑非常简单，没有什么需要解释的。不过设计文档和视图有些值得注意的地方。首先，设计文档里的`"language"`属性指明了处理文档的视图服务器。我们的代码是用 JavaScript 编写的，所以`"language"`属性的值也这样声明，如果不声明则默认用 JavaScript。如果你用的是 Erlang 或 Java，而不是 JavaScript，千万要记得指定。其次，所有 MapReduce 的视图代码都包含在`"view"`属性的值里。第三 MapReduce 键/值对里的键不必是字符串，可以是任何有效的 JSON 类型。计算每股每年最低价的视图从文档的 `date` 属性中提取出年份，用股票和年份作为键，从而简化了计算。第四，永久视图按 map 阶段生成的键对文档进行索引。如果生成的键是股票代号和年份，那么文档也会用这两个属性按给定次序索引。

可以通过访问视图来触发 MapReduce 运行。REST 风格的视图访问可以通过 Futon 控制台、通过 curl 等命令行工具或者其他支持 REST 交互的机制用浏览器来调用。

看过 MongoDB 和 CouchDB 这两种文档数据库使用 MapReduce 的例子后，下面介绍列族有序存储。

11.2　MapReduce 和 HBase

接下来我们加载 NYSE 数据集到 HBase 实例中。这一次用 MapReduce 来解析`.csv` 文件，并将数据加载到 HBase 中。这种“链式”MapReduce 用法非常流行，而且很适合解析大文件。数据加载入 HBase 以后，再用 MapReduce 来运行几个聚合函数。前面展示过 MapReduce 的两个例子。再来一个应该能帮助你加强对 MapReduce 概念的理解，同时展示它对多种情况的适应性。

在 HBase 中 Java 是 MapReduce 编程的首选。此外还可以用 Python、Ruby 或 PHP 编写 MapReduce 任务，把 HBase 当作任务的开始和/或终点。这个例子要创建以下四部分程序来协调工作。

- mapper 类生成键/值对。
- reducer 类接受 mapper 生成的值，通过处理创建聚合。在数据上传的例子里，mapper 只是插入数据到 HBase 表中。
- 驱动类连接 mapper 类和 reducer 类。
- 入口类在其 main 方法中触发任务。

这四部分也可以合并到一个类里面。这种情况下 mapper 类和 reducer 类可以改成静态嵌套类。不过在这个例子中我们还是创建四个类，每个类对应上面提到的一个程序部分。

假设 Hadoop 和 HBase 已经安装配置好了。请添加下列`.jar` 文件到 Java classpath 中，确保例子能编译通过和运行：

- `hadoop-0.20.2-ant.jar`

- hadoop-0.20.2-core.jar
- hadoop-0.20.2-tools.jar
- hbase-0.20.6.jar

hadoop jar 文件在 Hadoop 发行包里，hbase jar 则来自 HBase。

mapper 代码如下：

```
package com.treasuryofideas.hbasemr;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.MapWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class NyseMarketDataMapper extends
    Mapper<LongWritable, Text, Text, MapWritable> {

  public void map(LongWritable key, MapWritable value, Context context)
      throws IOException, InterruptedException {

    final Text EXCHANGE = new Text("exchange");
    final Text STOCK_SYMBOL = new Text("stockSymbol");
    final Text DATE = new Text("date");
    final Text STOCK_PRICE_OPEN = new Text("stockPriceOpen");
    final Text STOCK_PRICE_HIGH = new Text("stockPriceHigh");
    final Text STOCK_PRICE_LOW = new Text("stockPriceLow");
    final Text STOCK_PRICE_CLOSE = new Text("stockPriceClose");
    final Text STOCK_VOLUME = new Text("stockVolume");
    final Text STOCK_PRICE_ADJ_CLOSE = new Text("stockPriceAdjClose");

    try
    {
      //样例数据文件
      String strFile = "data/NYSE_daily_prices_A.csv";

      //create BufferedReader to read csv file
      BufferedReader br = new BufferedReader( new FileReader(strFile));
      String strLine = "";
      int lineNumber = 0;

     //按行读取数据
      while( (strLine = br.readLine()) != null)
      {
        lineNumber++;
                if(lineNumber > 1) {
                  String[] data_values = strLine.split(",");
                  MapWritable marketData = new MapWritable();
                  marketData.put(EXCHANGE, new Text(data_values[0]));
                  marketData.put(STOCK_SYMBOL, new Text(data_values[1]));
                  marketData.put(DATE, new Text(data_values[2]));
```

```
                    marketData.put(STOCK_PRICE_OPEN, new Text(data_values[3]));
                    marketData.put(STOCK_PRICE_HIGH, new Text(data_values[4]));
                    marketData.put(STOCK_PRICE_LOW, new Text(data_values[5]));
                    marketData.put(STOCK_PRICE_CLOSE, new Text(data_values[6]));
                    marketData.put(STOCK_VOLUME, new Text(data_values[7]));
                    marketData.put(STOCK_PRICE_ADJ_CLOSE, new Text(data_values[8]));

                    context.write(new Text(String.format("%s-%s", data_values[1],
data_values[2])), marketData);

                  }
      }


    }
    catch(Exception e)
    {
      System.errout.println("Exception while reading csv file or process
interrupted: " + e);
    }

    }
}
```

NyseMarketDataMapper.java

前面的代码非常基础，重点展示了 map 函数的关键功能。mapper 类继承自 org.apache.hadoop.mapreduce.Mapper，并实现了 map 方法。Map 方法接受键、值和上下文对象作为输入参数。注意，在 emit 方法中，我把股票代码和日期连在一起创建了一个复杂键。

.csv 解析逻辑本身很简单，可能只需稍作修改来支持条目内出现逗号的情况。不过对例子中这个数据集而言，它工作得很好。

第二部分是含有 reduce 方法的 reducer 类。reduce 方法把数据上传到 HBase 表中，代码如下：

```
public class NyseMarketDataReducer extends TableReducer<Text, MapWritable,
ImmutableBytesWritable> {
         public void reduce(Text arg0, Iterable arg1, Context context) {
        //股票代码和日期组成的复合键是唯一的，所以一个值对应一个键

        Map marketData = null;
        for (MapWritable value : arg1) {
            marketData = value;
            break;
        }

        ImmutableBytesWritable key = new ImmutableBytesWritable(Bytes
                .toBytes(arg0.toString()));
        Put put = new Put(Bytes.toBytes(arg0.toString()));
        put.add(Bytes.toBytes("mdata"), Bytes.toBytes("daily"), Bytes
                .toBytes((ByteBuffer) marketData));
        try {
```

```
            context.write(key, put);
        } catch (IOException e) {
            // TODO Auto-generated catch block
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
        }
    }

}
```

NyseMarketDataReducer.java

map 函数和 reduce 函数在驱动类里被连起来，代码如下：

```
public class NyseMarketDataDriver extends Configured implements Tool {
   @Override
    public int run(String[] arg0) throws Exception {
        HBaseConfiguration conf = new HBaseConfiguration();
        Job job = new Job(conf, "NYSE Market Data Sample Application");
        job.setJarByClass(NyseMarketDataSampleApplication.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setMapperClass(NyseMarketDataMapper.class);
        job.setReducerClass(NyseMarketDataReducer.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);

        FileInputFormat.addInputPath(job, new Path(
                "hdfs://localhost/path/to/NYSE_daily_prices_A.csv"));
        TableMapReduceUtil.initTableReducerJob("nysemarketdata",
                NyseMarketDataReducer.class, job);
        boolean jobSucceeded = job.waitForCompletion(true);
        if (jobSucceeded) {
            return 0;
        } else {
            return -1;
        }
    }

}
```

NyseMarketDataDriver.java

最后触发驱动，代码如下：

```
package com.treasuryofideas.hbasemr;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.util.ToolRunner;

public class NyseMarketDataSampleApplication {
    public static void main(String[] args) throws Exception {
        int m_rc = 0;
        m_rc = ToolRunner.run(new Configuration(),
new NyseMarketDataDriver(), args);
```

```
        System.exit(m_rc);
    }

}
```

NyseMarketDataSampleApplication.java

到这儿 MapReduce 和 HBase 的一个简单例子就介绍完了。下面你将会看到更多例子，它们比一句简单的 HBase 写操作更复杂。

11.3　MapReduce 和 Apache Mahout

MapReduce 可以用来解决很多问题。Google、Yahoo!、Facebook 和许多其他组织在各种各样丰富的场景中都使用了 MapReduce，这些场景包括分布式排序、Web 链接图遍历、日志文件统计、文档聚类和机器学习。不仅如此，MapReduce 能够适用的场景正变得越来越多。

Apache Mahout 期望用 Hadoop 和 MapReduce 建立起一套完整的可扩展的机器学习套件和数据挖掘类库。本节介绍 Mahout 和这个项目的一些例子。希望通过我的介绍能推动你进一步探索 MapReduce，帮助你在特定场景中高效地使用 MapReduce。

首先访问 mahout.apache.org 下载最新的发行包或者源代码。这个项目还在持续地快速演进和增加新功能，所以最好用源代码构建。除 JDK 外唯一需要的是 SVN 客户端和 Maven 3.0.2（或更高）版本，前一个用来下载源代码，后一个用来构建安装。

获取代码如下：

```
svn co http://svn.apache.org/repos/asf/mahout/trunk
```

然后进入源代码的“trunk”目录，执行下面的命令来编译和安装 Apache Mahout：

```
mvn compile
mvn install
```

还有 Mahout 样例：

```
cd examples
mvn compile
```

Mahout 附带了 taste-web recommender 样例程序。可以在 taste-web 目录下执行 `mvn` 来编译和运行程序。

虽然 Mahout 是一个新项目，但是它包含了聚类、分类、协同过滤和进化规划的实现。解释这些机器学习主题的含义超过了本书的范畴，不过我会介绍一个最基本的例子来展示如何使用 Mahout。

Mahout 包括了一个推荐引擎类库 Taste。它使用协同过滤，用来快速构建基于用户和基于物品的推荐系统。

Taste 主要包含以下五部分。

- DataModel：存储用户、物品和偏好的抽象模型。
- UserSimilarity：定义两个用户之间相似性的接口。

- ItemSimilarity：定义两个物品之间相似性的接口。
- Recommender：推荐提供者要实现的接口。
- UserNeighborhood：推荐系统用邻居表示相似用户，进而产生推荐。这个接口定义邻居。

你可以利用 Hadoop 在大数据集上运行批处理计算来构建推荐系统，使其成为高度可扩展的机器学习系统。

假设用户对一组物品的评分放在 `ratings.csv` 的文件里，每一行包括 `user_id`、`item_id`、`ratings`，这和前面的 MovieLens 数据集很相似。Mahout 有一套丰富的模型类来映射这个数据集。可以像下面这样使用 `FileDataModel`：

```
FileDataModel dataModel = new FileDataModel(new File(ratings.csv));
```

下面要确定一种测量距离的方法来表示两个不同用户的评分有多相似。欧几里德距离是这类方法中最简单的，很多时候也可以用皮尔森相关（Pearson correlation）。使用皮尔森相关，需配置相似性类型如下：

```
UserSimilarity userSimilarity = new PearsonCorrelationSimilarity(dataModel);
```

下面定义 UserNeighborhood 和 Recormmender，并结合它们来生成推荐。代码如下：

```
//Get a neighborhood of users
UserNeighborhood neighborhood =
        new NearestNUserNeighborhood(neighborhoodSize, userSimilarity, dataModel);
//Create the recommender
Recommender recommender =
        new GenericUserBasedRecommender(dataModel, neighborhood, userSimilarity);
User user = dataModel.getUser(userId);
System.out.println("User: " + user);
//Print out the users own preferences first
TasteUtils.printPreferences(user, handler.map);
//Get the top 5 recommendations
List<RecommendedItem> recommendations =
        recommender.recommend(userId, 5);
TasteUtils.printRecs(recommendations, handler.map);
```

'Taste' example

这就是构建并运行一个简单的推荐系统所需的所有工作。

上面的例子没有显式地使用 MapReduce，而是使用了基于协同过滤的推荐系统。Mahout 使用 MapReduce 来完成任务，利用 Hadoop 基础设施在大的分布式数据上计算推荐分数，只不过大部分底层基础设施被抽象掉了。

本章演示了一系列 MapReduce 样例，展示了如何能优雅地处理复杂的大数据集。不需要底层 API，无需担心资源死锁或者饥饿。此外保持数据和计算在一起减少了 I/O 和带宽约束的影响。

11.4 小结

MapReduce 是一种快速高效处理大量信息的方法。Google 用它来处理很多繁重的工作。

Google 非常善意地与研究和开发社区分享了它的底层思想。此外，Hadoop 团队构建了一个健壮的，可扩展的开源基础设施来利用这一处理模型。其他 NoSQL 项目和供应商也采用了 MapReduce。

在所有高可扩展和分布式模型中，MapReduce 都在代替 SQL 来处理海量数据。其性能和“无共享”模型成功地超越了传统 SQL 模型。

编写 MapReduce 程序相对简单，因为基础设施负责处理复杂性，开发者可以专注于链接 MapReduce 任务，并使用它们处理大量数据。常见的 MapReduce 任务可以交由一个公共的基础设施，比如 CouchDB 内建的 reducer，或者 Apache Mahout 这样的项目来处理。不过有时在定义键和使用 reducer 上还是需要多加注意。

第 12 章

使用 Hive 分析大数据

本章内容

- 介绍 Apache Hive，基于 Hadoop 的数据仓库
- 按样例学习 Hive
- 了解 Hive 命令语法和语义
- 用 Hive 查询 MovieLens 数据集

大数据核心问题的解决方案包括了宽松的数据结构、列族存储、分布式文件系统、复制，有时也包括最终一致性。这些解决方案的重点在于管理大量的、稀疏的、去正规化的数据，它们通常大到 TB 量级。使用这些存储分析和访问数据的方式一般都是特定的，或者预定义好的。因此即时查询和丰富的查询表达式的支持往往优先级并不高，而且也没有现成可用的方案。此外许多大数据解决方案使用的产品相对较新，仍在快速演进之中。这些产品还没有得到大范围测试，距离功能完善也很遥远。即是说，它们只擅长做被设计来做的事情：管理大数据。

相比新兴的大数据解决方案，RDBMS 有更成熟健壮的数据查询和管理工具。其中最显著、最重要的就是 SQL。作为查询数据的方式，它非常强大便利：可以分组、翻转、聚合和关联集合数据。NoSQL 中最为缺失的正是类似 SQL 的东西，这点尤为讽刺。

随着对类 SQL 语法和高层抽象之便利的需求逐渐觉醒，Hive 和 Pig 出现了。Apache Hive 是建立在 Hadoop 基础上的数据仓库，Apache Pig 是大数据分析的高级语言。本章介绍 Hive 和 Pig，并展示如何利用这些工具分析大数据集。

> Google App Engine（GAE）提供 GQL 来支持类 SQL 查询。

12.1 Hive 基础

开始学习 Hive 前，需要先安装 Hive。Hive 基于 Hadoop，因此要先安装 Hadoop。Hadoop 可以从 hadoop.apache.org 上下载（Hadoop 安装参阅附录 A）。目前 Hive 支持 Java 1.6 和 Hadoop 0.20.2，请注意获取这三个软件的正确版本。Hive 在 Mac OS X 和任何 Linux 版本上都没有问题。在 Windows 也许能通过 Cygwin 运行 Hive，但是本章不会介绍。如果你使用 Windows，没有

Mac OS X 或者 Linux 环境，可以考虑用 VMWare Player 和虚拟机。参阅附录 A 了解如何获取和安装虚拟机。

安装 Hive 很容易。只需执行下列步骤。

(1) 下载稳定版 Hive。Mac OS X 上下载 hive-0.6.0 使用命令 curl -O http://mirror.candidhosting.com/pub/apache//hive/hive-0.6.0/hive-0.6.0.tar.gz。Linux 上用 `wget` 代替 `curl`。

(2) 解压文件。在 Mac OS X 和 Linux 上，解压命令是 `tar zxvf hive-0.6.0.tar.gz`。

(3) 设置 `HIVE_HOME` 环境变量指向 Hive 安装目录。

(4) 添加`$HIVE_HOME/bin` 到 `PATH` 环境变量中，以方便在 Hive 主目录外使用。

(5) 在`$HADOOP_HOME` 目录中执行 `bin/start-all.sh`，以启动 Hadoop 后台进程。这样会启动 HDFS namenode、次级 namenode 和 datanode，还会启动 MapReduce 的 job tracker 和 task tracker。用 `jps` 命令确认这五个进程运行起来。

(6) 在 HDFS 上创建`/tmp` 和`/user/hive/warehouse` 目录，如下：

```
bin/hadoop fs -mkdir /tmp
bin/hadoop fs -mkdir /user/hive/warehouse
```

其中`/user/hive/warehouse` 是 `hive` 元存储仓库目录。

(7) 设置组用户对`/tmp` 和`/user/hive/warehouse` 目录的写权限。可以用 `chmod` 命令修改权限：

```
bin/hadoop fs -chmod g+w /tmp
bin/hadoop fs -chmod g+w /user/hive/warehouse
```

完成上述步骤后，就可以使用 Hive 上的 Hadoop 集群了。在`$HIVE_HOME` 目录下运行 `bin/hive` 来启动 Hive 命令行接口（CLI）。你会觉得使用 Hive CLI 的感觉似曾相识，因为其语法和命令行访问 RDBMS 的语法很相似。

> 在我的伪分布式本地安装里，`bin/start-all.sh` 会为 HDFS 和 MapReduce 守护进程产生 5 个 Java 进程。

开始先列出所有已存在的表：

```
SHOW TABLES;
```

hive_examples.txt

还没创建过表，所以会返回 `empty OK` 和执行耗时。大多数据库 CLI 都会输出查询耗时，它是查询高效与否的第一个指示器。

Hive 不是用来做实时查询的

Hive 在 Hadoop 上提供了优雅的类 SQL 查询框架。Hadoop 基础设施有很好的扩展性，能处理海量的分布式数据集。可以说，Hive 利用 HDFS 和 MapReduce 为查询和处理大数据集提供的抽象层很强大。

> 虽然如此，Hive 并不是实时查询系统。它更适合批量处理。Hive 对 Hadoop 和 MapReduce 的依赖导致它在任务提交与调度方面存在大量开销，因此 Hive 查询的延迟往往非常高。在你学习各种例子和使用 CLI 尝试 Hive 的过程中，就会注意到花费在执行查询上的时间，即便小数据集也在秒级，有时甚至是分钟级，这和 RDBMS 执行类似查询的耗时形成鲜明对比。Hive 中不存在查询缓存，所以重复查询耗时和第一次查询一样。
>
> 数据量变大时，Hadoop 在大规模上的效率让 Hive 的开销变得微不足道。当查询可能会扫描每行数据时，传统的 RDBMS 可能会回到表扫描。与之类似，在极大量数据和批量操作时，Hive 的性能最优。

下面，先创建一个表：

```
CREATE TABLE books (isbn INT, title STRING);
```

hive_examples.txt

我们创建了一个书籍表，包含两列：`isbn` 和 `title`。数据类型分别是整数和字符串。要显示表结构，查询如下：

```
hive> DESCRIBE books;
OK
Isbn     int
Title    string
Time taken: 0.263 seconds
```

hive_examples.txt

再创建一个 `users` 表：

```
CREATE TABLE users (id INT, name STRING) PARTITIONED BY (vcol STRING);
```

hive_examples.txt

`users` 表包含三列：`id`、`name` 和 `vcol`。你可以用 `DESCRIBE` 来确认表结构：hive> DESCRIBE users; OK Id int Name string Vcol string Time taken: 0.12 seconds hive_examples.txt

12

`vcol` 列是虚拟的，它表示分区，值取自数据被存储的分区，而非数据本身。一个表可以被划分成多个逻辑组成部分。每个逻辑组成部分有一个标识，标识存储在虚拟列中。

现在执行 `SHOW TABLES` 命令列出已创建的表：

```
hive> DESCRIBE users;
OK
Id      int
Name    string
Vcol    string
Time taken: 0.12 seconds
```

hive_examples.txt

表 book 存储图书数据，`isbn` 和 `title` 分别标识和描述一本书，但只有两个属性过于简单，加上 `author` 和 `category` 列似乎不错。在 RDBMS 里，这样的操作通过 `ALTER TABLE` 完成。

Hive 也支持类似的语法，这点也没啥奇怪的，可以像下面这样修改 book 表添加新列：

```
ALTER TABLE books ADD COLUMNS (author STRING, category STRING);
```

hive_examples.txt

重新确定 book 表修改后的结构：

```
hive> DESCRIBE books;
OK
Isbn   int
Title  string
Author string
Category     string
Time taken: 0.112 seconds
```

hive_examples.txt

下面修改 author 列来适应一书多作者的情况，这时字符串数组比单个字符串表达力更好。如果修改时还想在列上附加注释，说明此列存储多值数据，可以用下面的命令实现：

```
ALTER TABLE books CHANGE author author ARRAY<STRING> COMMENT "multi-valued";
```

hive_examples.txt

author 列的修改之后，重新运行 DESCRIBE TABLE，输出如下：

```
hive> DESCRIBE books;
OK
Isbn   int
Title  string
Author array<string>     multi-valued
Category     string
Time taken: 0.109 seconds
```

hive_examples.txt

ALTER TABLE 命令支持用下面的语法修改表属性：

```
ALTER TABLE table_name CHANGE [COLUMN]
old_column_name new_column_name column_type
[COMMENT column_comment]
[FIRST|AFTER column_name]
```

用 ALTER TABLE 时，命令参数必须和上面展示的顺序保持一致。方括号（[]）里的参数可选，剩下的必须以正确顺序出现在命令里。由此引出的副作用是，如果只想修改列的属性，而不是列名，那就要连续两次写出同一个列名。可以查看前面例子的 author 列来了解这是如何影响命令的。Hive 支持基本和复杂的数据类型。在 Hive 里复杂类型表示为映射表、数组或结构。前面例子里，列被修改为存储数组。数组元素需要额外的类型定义，在 author 列的例子中，ARRAY 里包含 STRING 类型。

下面，假设想要存储短故事、杂志等除书籍以外的出版物，就可能考虑用 published_contents 代替原来的表名。改表名可以这样做：

```
ALTER TABLE books RENAME TO published_contents;
```

hive_examples.txt

对 published_contents 执行 DESCRIBE TABLE 命令，输出如下：

```
hive> DESCRIBE published_contents;
OK
isbn    int
title   string
author  array<string>   multi-valued
category   string
Time taken: 0.136 seconds
```

hive_examples.txt

很明显，再对 books 执行 DESCRIBE TABLE 会返回错误：

```
hive> DESCRIBE books;
FAILED: Execution Error, return code 1 from org.apache.hadoop.hive.ql.exec.DDLTask
```

hive_examples.txt

下面再介绍一个更完整的样例来展示 Hive 的查询能力。因为 published_contents 和 users 表在本章接下来的部分中可能已经没用了，所以删除它们如下：

```
DROP TABLE published_contents;
DROP TABLE users;
```

12.2 回到电影评分

第 6 章里，我们学习了如何查询 NoSQL 存储的内容。那章用到了免费公开的电影评分数据来演示 NoSQL，特别是 MongoDB 的查询机制。现在让我们重新用 Hive 来处理这个数据集。继续之前回顾一下第 6 章的例子可能会有所帮助。

可以用下面的命令下载包含超过 100 万条评分的 MovieLens 数据集：

```
curl -O http://www.grouplens.org/system/files/million-ml-data.tar__0.gz
```

解压缩得到下面的文件：

- README
- movies.dat
- ratings.dat
- users.dat

ratings.dat 文件包含评分数据，每行一条评分记录。评分记录格式是这样的：UserID::MovieID::Rating::Timestamp。

> MovieLens 数据集里的评分、电影和用户数据用::分割。用 Hive 加载器根据这个分隔符解析加载数据时会遇到麻烦。所以我选择用#替换整个文件的所有::。就是用 vi 打开文件，然后替换所有出现的::分隔符，命令如下：
>
> ```
> :%s/::/#/g
> ```

12

修改完分隔符后将其保存到新文件中，每个文件后面加上.hash_delimited。这样就有了三个新文件：

- ❑ ratings.dat.hash_delimited
- ❑ movied.dat.hash_delimited
- ❑ users.dat.hash_delimited

我用新文件作数据源，原来的.dat 文件保持不动。

要把评分数据文件加载到同样结构的 Hive 表中，先得创建同样结构的 Hive 表：

```
hive> CREATE TABLE ratings(
    > userid INT,
    > movieid INT,
    > rating INT,
    > tstamp STRING)
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY '#'

    > STORED AS TEXTFILE;
OK
Time taken: 0.169 seconds
```

hive_movielens.txt

Hive 包含的工具支持用 `LOAD DATA` 命令从文件中加载数据。源可以是本地文件系统或者 HDFS，命令如下：

```
LOAD DATA LOCAL INPATH <'path/to/flat/file'> OVERWRITE INTO TABLE <table name>;
```

加载时不会执行任何验证。因此开发者要确保文件的数据格式匹配表结构。语法支持指定本地文件系统或 HDFS。基本上，在 `LOAD DATA` 后面声明 `LOCAL` 会指定源是本地文件系统。不包含 `LOCAL` 则意味着数据在 HDFS 上。如果文件在 HDFS 上，数据就只是拷贝到 Hive 的 HDFS 命名空间里，执行的是 HDFS 移动命令，所以它比从本地文件系统加载数据要快多了。数据加载命令还允许覆写数据或者将其追加到已有表中。命令中存在 `OVERWRITE` 和不存在 `OVERWRITE` 分别指覆写和追加。

我们已经下载了 MovieLen 数据。再准备一份拷贝，用#号替换掉分隔符::。然后将准备好的数据加载到 Hive 的 HDFS 命名空间中，加载命令如下：

```
hive> LOAD DATA LOCAL INPATH '/path/to/ratings.dat.hash_delimited'
    > OVERWRITE INTO TABLE ratings;
Copying data from file:/path/to/ratings.dat.hash_delimited
Loading data to table ratings
OK
Time taken: 0.803 seconds
```

hive_movielens.txt

加载入 Hive 表的评分记录超过了 100 万条。可以用你所熟悉的 `SELECT COUNT` 确认一下：

```
hive> SELECT COUNT(*) FROM ratings;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
Starting Job = job_201102211022_0012, Tracking URL =
http://localhost:50030/jobdetails.jsp?jobid=job_201102211022_0012
Kill Command = /Users/tshanky/Applications/hadoop/bin/../bin/hadoop job  -
Dmapred.job.tracker=localhost:9001 -kill job_201102211022_0012
2011-02-21 15:36:50,627 Stage-1 map = 0%,  reduce = 0%
2011-02-21 15:36:56,819 Stage-1 map = 100%,  reduce = 0%
2011-02-21 15:37:01,921 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201102211022_0012
OK
1000209
Time taken: 21.355 seconds
```

hive_movielens.txt

输出确认表里的记录超过了100万条。查询机制确认：在Hive里SQL计数的老方法还是管用的。在计数的例子里，我特地把 `SELECT COUNT` 所有控制台输出都包括进来，以便提示你注意以下一些重要的地方。

- Hive操作翻译成MapReduce任务。
- Hive操作的延迟很高。总共耗费21.355秒来执行计数。立即重跑也不会缩短耗时。再跑一次的耗时大致相同，因为不存在查询缓存机制。

Hive支持非常全面的过滤和聚合查询。你可以用 `WHERE` 子句来过滤数据。结果也可以使用 `GROUP BY` 分组。`DISTINCT` 参数可以帮助列出不重复的值。两个表可以用 `JOIN` 操作来合并。此外还可以编写自定义脚本来处理数据，并将其传入map和reduce函数。

为了进一步深入了解Hive的能力和它强大的查询机制，我们把MovieLens数据里的电影和用户数据也装入对应的表中，这会为探索Hive的功能提供一个绝佳的样例集。电影数据集的每行格式是MovieID::Title::Genres。其中 `MovieID` 是整数，`Title` 是字符串，`Genres` 也是字符串。`Genres` 字符串包括多个值，用管道符间隔。我们先创建 `movies` 表：

> 和评分数据一样，movies.dat 里原本的分隔符::被改成了#。

```
hive> CREATE TABLE movies(
    > movieid INT,
    > title STRING,
    > genres STRING)
    > ROW FORMAT DELIMITED
```

```
    > FIELDS TERMINATED BY '#'
    > STORED AS TEXTFILE;
OK
Time taken: 0.075 seconds
```

hive_movielens.txt

加载 movies 表数据如下：

```
hive> LOAD DATA LOCAL INPATH '/path/to/movies.dat.hash_delimited'
    > OVERWRITE INTO TABLE movies;
```

字符串 genres 包含多个值，例如这样：Animation|Children's|Comedy。因此这个数据存储为 ARRAY 而非 STRING 可能更好。存储成 ARRAY 更方便在查询参数中使用这些值。Hive 可以接受集合和映射表的分隔符参数，所以能很容易地分割 genres 并存储。修改后的 CREATE TABLE 和 LOAD DATA 命令如下：

```
hive> CREATE TABLE movies_2(
    > movieid INT,
    > title STRING,
    > genres ARRAY<STRING>)
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY '#'
    > COLLECTION ITEMS TERMINATED BY '|'
    > STORED AS TEXTFILE;
OK
Time taken: 0.037 seconds
hive> LOAD DATA LOCAL INPATH '/path/to/movies.dat.hash_delimited'
    > OVERWRITE INTO TABLE movies_2;
Copying data from file:/path/to/movies.dat.hash_delimited
Loading data to table movies_2
OK
Time taken: 0.121 seconds
```

hive_movielens.txt

加载完数据以后，用 SELET 打印出一些记录，用 LIMIT 限制结果集为五条记录：

```
hive> SELECT * FROM movies_2 LIMIT 5;
OK
1       Toy Story (1995)      ["Animation","Children's","Comedy"]
2       Jumanji (1995)        ["Adventure","Children's","Fantasy"]
3       Grumpier Old Men (1995)      ["Comedy","Romance"]
4       Waiting to Exhale (1995)     ["Comedy","Drama"]
5       Father of the Bride Part II (1995)      ["Comedy"]
Time taken: 0.103 seconds
```

hive_movielens.txt

MovieLens 的第 3 个数据集是 users.dat。其中每行的格式如下：UserID::Gender::Age::Occupation::Zip-code。下面是一行样例：

```
1::F::1::10::48067
```

性别（Gender）、年龄（Age）和职业（Occupation）都是离散值。性别是男或女，分别表示

为 M 和 F。年龄可以表示成递增函数和区间最小值，所有年龄都四舍五入到最近的年份里，且范围是开区间。职业是映射到字符值的离散数字。职业有下面这 20 个可能的值：

- 0：other or not specified
- 1：academic/educator
- 2：artist
- 3：clerical/admin
- 4：college/grad student
- 5：customer service
- 6：doctor/health care
- 7：executive/managerial
- 8：farmer
- 9：homemaker
- 10：K-12 student
- 11：lawyer
- 12：programmer
- 13：retired
- 14：sales/marketing
- 15：scientist
- 16：self-employed
- 17：technician/engineer
- 18：tradesman/craftsman
- 19：unemployed
- 20：writer

存储职业的字符串而不是数值可能更好，因为这样的话，浏览数据时就更容易理解其中的含义。要实现这一点，可以通过数据加载操作的关联脚本实现。Hive 对 map 和 reduce 函数支持可插拔的外部脚本。对 map 和 reduce 函数插入外部脚本跟在 Hive 表之间相互复制数据有关，如图 12-1 所示。

为了实际看到外部脚本的效果，特别是将 users 表的职业代号替换为字符串的脚本，必须首先创建 users 表并载入数据。创建 users 表命令如下：

```
hive> CREATE TABLE users(
    > userid INT,
    > gender STRING,
    > age INT,
    > occupation INT,
    > zipcode STRING)
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY '#'
    > STORED AS TEXTFILE;
```

hive_movielens.txt

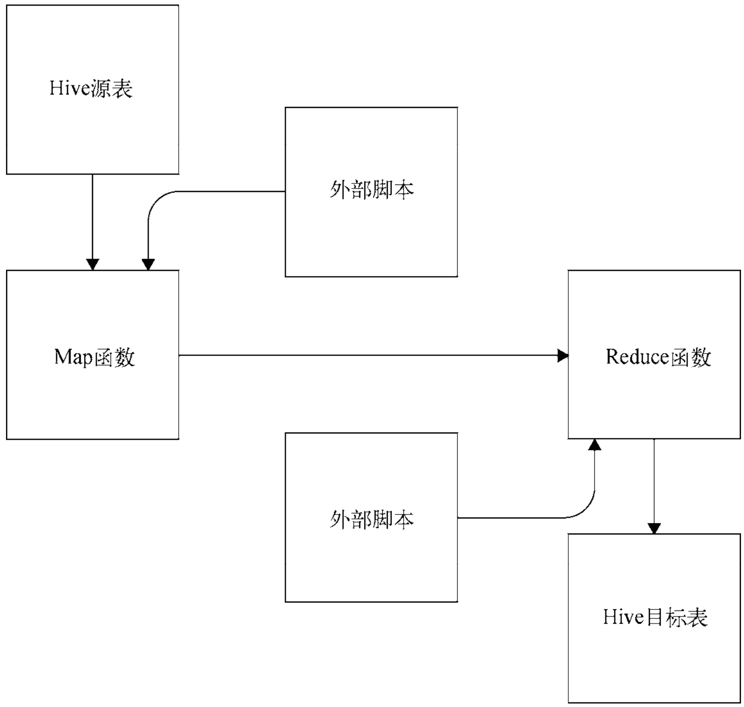


图　12-1

加载数据到表中如下：

```
hive> LOAD DATA LOCAL INPATH '/path/to/users.dat.hash_delimited'
    > OVERWRITE INTO TABLE users;
```

下面创建第 2 个 users 表 users_2，然后将 users 表中的数据灌进第 2 个表里。灌数据的过程中，利用外部脚本 occupation_mapper.py 把职业的整数值映射成对应的字符串值，然后把字符串灌进 users_2。数据转换代码如下：

```
hive> CREATE TABLE users_2(
    > userid INT,
    > gender STRING,
    > age INT,
    > occupation STRING,
    > zipcode STRING)
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY '#'
    > STORED AS TEXTFILE;
OK
Time taken: 0.359 seconds
hive> add FILE
/Users/tshanky/workspace/hadoop_workspace/hive_workspace/occupation_mapper.py;
hive> INSERT OVERWRITE TABLE users_2
    > SELECT
    > TRANSFORM (userid, gender, age, occupation, zipcode)
```

```
> USING 'python occupation_mapper.py'
> AS (userid, gender, age, occupation_str, zipcode)
> FROM users;
```

hive_movielens.txt

occupation_mapper.py 脚本如下：

```
occupation_dict = { 0:  "other or not specified",
  1:  "academic/educator",
  2:  "artist",
  3:  "clerical/admin",
  4:  "college/grad student",
  5:  "customer service",
  6:  "doctor/health care",
  7:  "executive/managerial",
  8:  "farmer",
  9:  "homemaker",
  10:  "K-12 student",
  11:  "lawyer",
  12:  "programmer",
  13:  "retired",
  14:  "sales/marketing",
  15:  "scientist",
  16:  "self-employed",
  17:  "technician/engineer",
  18:  "tradesman/craftsman",
  19:  "unemployed",
  20:  "writer"
}
for line in sys.stdin:
  line = line.strip()
  userid, gender, age, occupation, zipcode = line.split('#')
  occupation_str = occupation_map[occupation]
  print '#'.join([userid, gender, age, occupation_str, zipcode])
```

occupation_mapper.py

12

脚本含义非常清楚，用职业代号查找 occupation_dict 字典，将 users 表里每个职业代号替换成了对应的字符串值。

数据准备好以后，就可以使用亲切的 SQL 来查询了。

12.3 亲切的 SQL

SQL 有许多很好的特性，其中用 WHERE 子句过滤数据可能是最常用、最受欢迎的特性了。本节你将会看到 Hive 是如何支持 WHERE 子句的。

首先，从 movies 表中任取 5 部电影。可以用 LIMIT 限制只取 5 条记录：

```
SELECT * FROM movies LIMIT 5;
```

hive_movielens.txt

在我这儿 5 条记录分别是：

```
1      Toy Story (1995)    Animation|Children's|Comedy
2      Jumanji (1995)     Adventure|Children's|Fantasy
3      Grumpier Old Men (1995)     Comedy|Romance
4      Waiting to Exhale (1995)     Comedy|Drama
5      Father of the Bride Part II (1995)     Comedy
```

要使用 HiveQL（Hive 查询语言）列出所有与 Toy Story(1995)（电影 ID 为 1）的评分，可以查询如下：

```
hive> SELECT * FROM ratings
    > WHERE movieid = 1;
```

hive_movielens.txt

电影 ID 是数字，要计算所有 ID 小于 10 的电影的评分记录总数，可以用 HiveQL 如下：

```
hive> SELECT COUNT(*) FROM ratings
    > WHERE movieid < 10;
```

hive_movielens.txt

MapReduce 任务的输出是 5290。

要找出有多少用户对电影 Toy Story (1995)给出了 5 分评价，可以查询如下：

```
hive> SELECT COUNT(*) FROM ratings
    > WHERE movieid = 1 and rating = 5;
```

hive_movielens.txt

这个例子展示了在 `WHERE` 子句中使用超过 1 个条件的情况。在 `SELECT` 语句中可以使用 `DISTINCT` 来获取唯一值，默认是返回重复值。

Hive 中没有用来支持模糊匹配的 `LIKE`。不过 `SELECT` 语句支持在 `WHERE` 子句中结合列名使用正则表达式。要找出所有影片名以 `Toy` 开头的电影，可以这样查询：

```
hive> SELECT title FROM movies
    > WHERE title = `^Toy+`;
```

hive_movielens.txt

注意正则表达式声明在反括号里。这里的正则表达式遵从 Java 正则表达式语法。正则表达式还可以用来指定返回特定列。例如，可以只返回那些以字符 ID 结尾的列，像这样：

```
hive> SELECT `*+(id)` FROM ratings
    > WHERE movieid = 1;
```

hive_movielens.txt

表 `ratings` 包含电影的评分值。评分值可以是 1 到 5 之间的任意数字。如果想找出电影 Toy Story(1995)（`movieid = 1`）不同评分的个数，可以用 `GROUP BY` 查询如下：

```
hive> SELECT ratings.rating, COUNT(ratings.rating)
    > FROM ratings
    > WHERE movieid = 1
    > GROUP BY ratings.rating;
```

hive_movielens.txt

输出如下：

```
1      16
2      61
3      345
4      835
5      820
Time taken: 24.908 seconds
```

一个查询里可以包含多个聚合函数，比如 count、sum 和 average，只要所有操作都针对同一列即可。同一个查询里不能在多个列上执行聚合函数。

要在 map 层面上执行聚合，可以设置 `hive.map.aggr` 为 `true` 然后运行查询：

```
set hive.map.aggr=true;
SELECT COUNT(*) FROM ratings;
```

HiveQL 还支持用 `ORDER BY` 对结果集按升序或降序排列。要获取 `movies` 表里的所有记录并按 `movieid` 降序排列，可以这样查询：

```
hive> SELECT * FROM movies
    > ORDER BY movieid DESC;
```

hive_movielens.txt

Hive 还有另外一个排序工具 `SORT BY`，与 `ORDER BY` 相似之处是它也对记录按升序或降序排列。与 `ORDER BY` 不同的是，`SORT BY` 在 reducer 层面上实施排序。这意味着最终的结果可能只是部分有序。同一个 reducer 上所有记录会整体排序，但是多个 reducer 的记录在一起就不是这样了。

Hive 支持按虚拟列对数据集分区。可以用 `DISTRIBUTE BY` 把不同分区的数据发到不同的 reducer 上。分发到不同 reducer 上的数据可以在 reducer 层面上排序。`DISTRIBUTE BY` 和 `ORDER BY` 可以简写成 `CLUSTER BY`。

对那些熟悉 RDBMS 和 SQL，又很想探索 Hadoop 大数据，还希望能使用熟悉工具的开发者而言，HiveQL 的类 SQL 语法非常友好。探索 Hive 的 SQL 开发者很快就会开始盼望这个有力的工具：SQL 连接。Hive 甚至在这一点上也不会让人失望。HiveQL 支持连接。

12.4 HiveQL 连接

Hive 支持内连接、外连接和左外连接。要获取电影评分和影片名称，可以连接 `ratings` 表和 `movies` 表，查询如下：

```
hive> SELECT ratings.userid, ratings.rating, ratings.tstamp, movies.title
    > FROM ratings JOIN movies
    > ON (ratings.movieid = movies.movieid)
    > LIMIT 5;
```

输出如下：

```
376     4     980620359     Toy Story (1995)
1207    4     974845574     Toy Story (1995)
28      3     978985309     Toy Story (1995)
193     4     1025569964    Toy Story (1995)
1055    5     974953210     Toy Story (1995)
Time taken: 48.933 seconds
```

连接不限于两张表，可以更多。要获取电影评分、影片名称和用户性别（对电影给出评分的人的性别），可以连接 ratings、movies 和 users 表，查询如下：

```
hive> SELECT ratings.userid, ratings.rating, ratings.tstamp, movies.title,
 users.gender
    > FROM ratings JOIN movies ON (ratings.movieid = movies.movieid)
    > JOIN users ON (ratings.userid = users.userid)
    > LIMIT 5;
```

输出如下：

```
1  3  978300760  Wallace & Gromit: The Best of Aardman Animation (1996)  F
1  5  978824195  Schindler's List (1993)  F
1  3  978301968  My Fair Lady (1964)  F
1  4  978301398  Fargo (1996)  F
1  4  978824268  Aladdin (1992)  F
Time taken: 84.785 seconds
```

数据是有序的，所以会先接受到所有女性（F）相关的数据。如果想只获取男性用户记录，可以修改查询增加 WHERE 子句如下：

```
hive> SELECT ratings.userid, ratings.rating, ratings.tstamp, movies.title,
users.gender
    > FROM ratings JOIN movies ON (ratings.movieid = movies.movieid)
    > JOIN users ON (ratings.userid = users.userid)
    > WHERE users.gender = 'M'
    > LIMIT 5;
```

输出如下：

```
2  5  978298625  Doctor Zhivago (1965)  M
2  3  978299046  Children of a Lesser God (1986)  M
2  4  978299200  Kramer Vs. Kramer (1979)  M
2  4  978299861  Enemy of the State (1998)  M
2  5  978298813  Driving Miss Daisy (1989)  M
Time taken: 80.769 seconds
```

Hive 还支持更多的类 SQL 功能，包括 UNION 和子查询。举个例子，可以用 UNION 操作合并两个结果集如下：

```
select_statement UNION ALL select_statement UNION ALL select_statement ...
```

进一步查询并过滤 SELECT 语句合并的结果，可能的 SELECT 语句像下面这样：

```
SELECT *
FROM (
  select_statement
  UNION ALL
  select_statement
) unionResult
```

Hive 还支持 FROM 子句里的查询。比如要获取所有用户，他们为超过 15 部电影给出 5 分的最好评分：

```
hive> SELECT user_id, rating_count
    > FROM (SELECT ratings.userid as user_id, COUNT(ratings.rating) as
rating_count
    > FROM ratings
    > WHERE ratings.rating = 5
    > GROUP BY ratings.userid ) top_raters
    > WHERE rating_count > 15;
```

除了已经演示过的部分，还有很多与 Hive 及其查询语言有关的内容。不过在这里画上句号，做个总结可能更好。从本章介绍的内容可以看出，Hive QL 与 SQL 类似，有助于那些刚开始用 NoSQL 的开发者填补感觉上的空白。Hive 提供了足够好的抽象，使得开发者能够很容易地进行大数据处理。

结束本章之前，为了完整起见还需要介绍一些内容。首先简短地介绍一下计划解释，它有助于理解查询背后的 MapReduce。然后用一个小例子来介绍数据分区。

12.4.1　计划解释

绝大部分 RDBMS 都包含一个用来解释查询细节的工具。它们通常会详述这些方面：索引使用情况、访问到的记录、每个步骤的耗时等。Hadoop 是利用 MapReduce 进行分布式大规模处理的批处理系统。Hive 基于 Hadoop 并利用 MapReduce。Hive 里的计划解释可以揭示出查询背后的 MapReduce。

一个简单的例子可能是下面这样的：

```
hive> EXPLAIN SELECT COUNT(*) FROM ratings
    > WHERE movieid = 1 and rating = 5;
OK
ABSTRACT SYNTAX TREE:
  (TOK_QUERY (TOK_FROM (TOK_TABREF ratings))
(TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE))
(TOK_SELECT (TOK_SELEXPR (TOK_FUNCTIONSTAR COUNT)))
(TOK_WHERE (and (= (TOK_TABLE_OR_COL movieid) 1)
(= (TOK_TABLE_OR_COL rating) 5)))))
STAGE DEPENDENCIES:
  Stage-1 is a root stage
  Stage-0 is a root stage

STAGE PLANS:
  Stage: Stage-1
    Map Reduce
      Alias -> Map Operator Tree:
```

12

```
          ratings
            TableScan
              alias: ratings
              Filter Operator
                predicate:
                    expr: ((movieid = 1) and (rating = 5))
                    type: boolean
                Filter Operator
                  predicate:
                      expr: ((movieid = 1) and (rating = 5))
                      type: boolean
                  Select Operator
                    Group By Operator
                      aggregations:
                            expr: count()
                      bucketGroup: false
                      mode: hash
                      outputColumnNames: _col0
                      Reduce Output Operator
                        sort order:
                        tag: -1
                        value expressions:
                              expr: _col0
                              type: bigint
        Reduce Operator Tree:
          Group By Operator
            aggregations:
                  expr: count(VALUE._col0)
            bucketGroup: false
            mode: mergepartial
            outputColumnNames: _col0
            Select Operator
              expressions:
                    expr: _col0
                    type: bigint
              outputColumnNames: _col0
              File Output Operator
                compressed: false
                GlobalTableId: 0
                table:
                    input format: org.apache.hadoop.mapred.TextInputFormat
                    output format:
  org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat

    Stage: Stage-0
      Fetch Operator
        limit: -1


  Time taken: 0.093 seconds
```

如果还需要有关物理文件的信息，可以在 EXPLAIN 和查询之间加上 EXTENDED。下面介绍一个数据分区的简单例子。

12.4.2　分区表

表分区可以将数据分到多个命名空间中去，并基于命名空间标识符对数据进行过滤和查询。比方说，假设分析师相信评分会受到用户提交评分时间的影响，因此想将评分划为两部分，一份包括晚上 8 点到早上 8 点间的评分，另一份包括剩下时间段的评分。此时就可以创建虚拟列来标识出这个分区，并依此保存数据。

这样就可以按这些命名空间进行过滤、搜索和聚类了。

12.5　小结

本章简要描绘了 Hive 的强大和灵活性。展示了 SQL 如何与 Hadoop 结合起来提供一个引人注目的数据分析工具，一个传统 RDBMS 的开发者和新兴大数据开拓者都可以使用的工具。

Hive 由 Facebook 开发，并开源作为 Hadoop 的一个子项目。它现在已经成为顶级项目，并继续处于快速发展中，填补着 SQL 和 NoSQL 之间的空白。在 Hive 开源之前，有争议认为 Hadoop 只对那些需要访问“大数据”的团队的部分开发者有用。有人说 Hive 让 NoSQL 这个词不再有效果。可以说 Hive 恰好对此类质疑作出了强有力的声明：NoSQL 其实是 Not Only SQL 的缩写。

第 13 章

综览数据库内部

本章内容

- 了解 MongoDB、Membase、Hypertable、Apache Cassandra 和 Berkeley DB 的内部
- 了解一些 NoSQL 产品内部架构
- 理解内部设计选择

学习一种产品/工具至少涉及下面三个维度：

- 理解语法语义
- 通过尝试和使用来学习
- 理解内部构造，知道底层发生了什么

前面章节里，我们学习了很多数据库的使用，特别是 MongoDB、Redis、CouchDB、HBase 和 Cassandra。这许多例子展示了语法，也解释了概念，动手尝试这些例子对学习 NoSQL 产品有很好的帮助。之前我们偶尔会了解一些底层。本章我们将深入浏览 NoSQL 产品的架构和内部结构。和其他章节一样，我会选择一组不同类型的 NoSQL 产品，本章选择这些产品的原因列举一部分如下：

- MongoDB。本书的很多章节中都讲到了 MongoDB，也介绍过 MongoDB 内部的许多方面，更多介绍能帮助你全面了解这个产品。
- Membase。但凡提到基于内存，同时还提供磁盘持久化的键/值存储，Redis 就是典范。Redis 的功能和内部结构在多个章节里说明过。本章将跳过 Redis，介绍另一种具有竞争力的产品 Membase。Membase 由于其显著的性能特点得到广泛使用，由于采用了 Memcached 协议，它能够成为 Memcached 的替代品。
- Hypertable。提到有序列族存储，我们就会说起 HBase。HBase 非常类似 Google Bigtable，而且很受欢迎，但还有一些产品也构建在同样的模型上。这些产品包括 Hypertable 和 Cloudata。Cloudata 是更新的开源产品，而 Hypertable 则是更完善的选择。许多大型互联网应用和服务都部署 Hypertable 作为可扩展的数据存储。Hypertable 用 C++写成，与 HBase 相比，其性能更好。所以本章会介绍 Hypertable 的许多方面。
- Apache Cassandra。Google Bigtable 和 Amazon Dynamo 是构建大规模 NoSQL 存储的两种流行范本。Apache Cassandra 博采众长，结合了这二者。我们讨论过 Apache Cassandra，它的快速写能力也非常出色。本章介绍将 Cassandra 的一些组成结构。

- Berkeley DB。Berkeley DB 是非常强大的键/值存储，一些前沿 NoSQL 产品，例如 Amazon Dynamo、LinkedIn、Voldemort 和 GenieDB 用它做底层存储。

本章并非面面俱到，也并非适用于所有产品。不过我希望这些内容能引发你的兴趣，去进一步了解 NoSQL 产品的结构，去阅读 NoSQL 产品的代码，从使用这些产品中受益，并且有机会为 NoSQL 做出自己的贡献，使它们变得更好。另一方面，本章提到的一些 NoSQL 产品在其他章节只是一笔带过，有些甚至根本不会再提到，通过这一章的介绍，希望能激发你探索这部分 NoSQL 产品的兴趣。

13.1 MongoDB 内部

目前为止，我们已经介绍过了 MongoDB 的命令和使用，了解了它的存储和查询机制，并在这个基础上学习了有关复制的内容，还学习了存储格式 BSON 的一些细节。本节，我们再介绍 MongoDB 的一些重要内容，这些内容是以你之前的了解为基础的。

MongoDB 遵从客户端–服务器架构，这种架构在传统 RDBMS 中也很常见。客户端–服务器架构包含一个服务器和多个连接到服务器上的客户端。如果使用分片和复制，则用多个服务器来代替单个服务器。无论是单个节点还是采用分片和复制，数据都是在客户端和服务器之间，以及节点之间，来回传送。

BSON 规范

MongoDB 将其文档编码为类 JSON 的二进制格式 BSON，这个格式第 2 章里介绍过。在此简要复习一下 JSON 的一些特性。

一个 BSON 文档是零或更多个二进制键/值对组成的集合，其中基本二进制类型如下。

- Byte：1 字节
- int32：4 字节
- int64：8 字节
- double：8 字节

int32 和 int64 分别对应 32 位和 64 位有符号整数。Double 对应 64 位 IEEE 754 浮点值。下面是一个文档的例子：

```
{ "hello": "world" }
```

这个文档表示为如下的 BSON 格式：

```
"\x16\x00\x00\x00\x02hello\x00 \x06\x00\x00\x00world\x00\x00"
```

这里展示的 BSON 符号使用了我们熟悉的 C 的 16 进制表示。如果想把二进制形式和文档对应起来，其实是这样的：

```
"{ and }—"\x16\x00\x00\x00 and \x00
"hello":—x02hello\x00
"world"—\x06\x00\x00\x00world\x00
```

阅读更多有关 BSON 规范的内容请参阅 http://bsonspec.org/。

13.1.1　MongoDB传输协议

客户端通过基于 TCP/IP 的套接字连接与 MongoDB 服务器通讯。使用的传输协议是一种简单的请求–响应式套接字协议。传输协议头部和主体都用 BSON 编码，消息采用小端字节序（little endian），和 BSON 一样。

在标准的请求–响应模型中，客户端发送请求给服务器，服务器对请求作出响应。从传输协议来看，请求包含报头和请求主体，响应则包含报头和响应主体。请求和响应的报头格式相近，消息主体格式却各有不同。图 13-1 展示了客户端和 MongoDB 服务器之间基本的请求-响应。

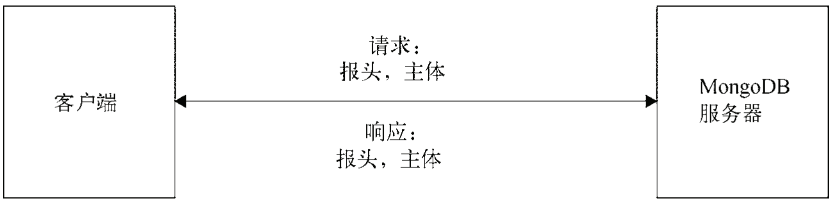


图　13-1

MongoDB 传输协议支持很多操作，如下面这些。

> RESERVED 也是操作，原来用于 OP_GET_BY_OID。没有把它列进来是因为用得不多。

- OP_INSERT（代码：2002）：插入文档，CRUD 之“创建”操作。

> CRUD 分别表示创建（Create）、读取（Read）、更新（Update）和删除（Delete），是标准的数据管理操作，许多与数据交互的系统和接口都支持 CRUD 操作。

- OP_UPDATE（代码：2001）：更新文档，CRUD 中的更新操作。
- OP_QUERY（代码：2004）：查询集合文档，CRUD 中的读取操作。
- OP_GET_MORE（代码：2005）：获取同一个查询的更多数据。查询的响应结果可以包含大量文档，为了提高性能，同时避免发送整个文档集，数据库用游标支持增量获取记录。OP_GET_MORE 便于通过游标获取更多的文档。
- OP_REPLY（代码：1）：应答客户端请求，响应 OP_QUERY 和 OP_GET_MORE 操作。
- OP_KILL_CURSORS（代码：2007）：关闭游标。
- OP_DELETE（代码：2006）：删除文档。
- OP_MSG（代码：1000）：通用消息命令。

所有请求响应都包含报头，标准的报头包含下列属性。

- messageLength：消息长度，单位是字节，包括保存消息长度所需的 4 字节。

- requestID：唯一的消息标识符，由发起操作的客户端或服务器生成。
- responseTo：数据库对 OP_QUERY 和 OP_GET_MORE 的响应中，将客户端请求的 requestID 存储在 responseTo 里，这样客户端就能匹配请求和响应。
- opCode：操作代码，支持的操作列在本节前文中。

下面了解几个简单而且常见的请求–响应场景。

13.1.2　插入文档

创建和插入新文档时，客户端通过请求发送 OP_INSERT 操作，其中包括以下内容。

- **消息头**：标准消息头结构，包括 messageLength、requestID、responseTo 和 opCode。
- **1 个 int32 整数值**：0（保留将来使用）。
- **1 个字符串**：完整的集合名。例如数据库 aDatabase 的集合 aCollection 显示为 aDatabase.aCollection。
- **1 个数组**：数组包含 1 个或多个需要被插入集合的文档。

数据库处理插入文档的请求，可以通过 getLastError 命令检查请求结果。然而，对插入文档的请求，数据库不会显式响应。

13.1.3　查询集合

查询时，客户端通过请求发送 OP_QUERY 操作，数据库返回的响应中包含相关文档集合。

客户端发送的 OP_QUERY 消息包括以下内容。

- **消息头**：标准消息头结构，包括 messageLength、requestID、responseTo 和 opCode。
- **一个 int32 整数值**：标志位，表示查询选项。标志定义了游标、结果流和部分结果（如果有分片宕机了）的属性。例如返回数据后游标是否关闭，或者游标空闲一段时间后是否超时。
- **一个字符串**：完整集合名。
- **一个 int32 整数值**：跳过的文档数量。
- **又一个 int32 整数值**：要返回的文档数量。数据库会对应接受文档的请求以一个 OP_REPLY 操作作为响应。如果文档总数大于返回的文档数，通常会返回一个游标。受驱动及其限制结果集的能力影响，这个属性的取值会不同。
- **BSON 格式的查询文档**：被搜索的文档要匹配的查询条件。
- **一个文档**：要返回的字段，表现为 BSON 格式。

对客户端 OP_QUERY 请求，MongoDB 服务器会返回 OP_REPLY，其内容包括下面部分。

- **消息头**：客户端请求和服务器响应的消息头非常相似。另外前面提到过，OP_REPLY 的 responseTo 属性会包含客户端请求 OP_QUERY 的 requestID。
- **一个 int32 整数值**：响应标志，通常说明存在错误或异常，可能包含有关查询错误或出错游标标识符的信息。

13

- **一个 int64**：游标标识符，客户端通过标识符可以获取更多文档。
- **一个 int32 整数值**：游标起始点。
- **又一个 int32 整数值**：返回的文档的数量。
- **一个数组**：查询结果文档。

这只是传输协议的一部分。更多有关传输协议的内容请参阅：www.mongodb.org/display/DOCS/Mongo+Wire+Protocol。也可以浏览 MongoDB 源代码，地址：https://github.com/mongodb。

总之，文档全部存在服务器端。客户端通过服务器来执行文档插入、更新和文档。客户端和服务器之间通信使用高效的二进制格式和传输协议。下面谈谈存储架构。

13.1.4　MongoDB数据库文件

MongoDB 把数据库和集合的数据存储在文件中，文件放在 `mongod --dbpath` 选项指定路径下。`dbpath` 默认是`/data/db`。MongoDB 文档存储成文件时遵循一定的结构。下面先介绍如何查询集合的存储属性，然后介绍文件分配方式的细节。

通过 Mongo shell 可以查询集合存储属性。启动 mongod，用命令行程序连上服务器。然后可以像下面这样查询集合大小：

```
> db.movies.dataSize();
327280
```

mongodb_data_size.txt

这个例子用了第 6 章的`movies`集合。文件`movies.dat`包含电影评分数据集中的电影信息，共 171 308 字节，但是对应的集合大很多，因为 MongoDB 格式还要另外存储元数据。返回的大小不是磁盘上实际存储的大小，只是数据的大小。为集合分配的存储空间可能包含未被使用的空间。要获取集合实际存储的大小，查询如下：

```
> db.movies.storageSize();
500480
```

mongodb_data_size.txt

实际存储大小为 500 480 字节，相比之下数据就小多了，只有 327 280 字节。这个集合可能还有索引。要查询集合的总大小，包括数据、未分配的存储和索引，可以这样查询：

```
> db.movies.totalSize();
860928
```

mongodb_data_size.txt

为了确认三个不同的值能对得上，需要找出索引占用的空间。这就需要`movies`集合相关索引的集合名。要找出`movies`集合索引的全称，可以像下面这样查询系统中所有命名空间：

```
> db.system.namespaces.find()
```

mongodb_data_size.txt

我的 MongoDB 实例里有很多集合，列表很长，和这个例子有关的信息是这些：

```
{ "name" : "mydb.movies" }
{ "name" : "mydb.movies.$_id_" }
```

mydb.movies 是集合本身，另一个 mydb.movies.$_id_是 id 上索引的集合。要查看索引集合的数据大小、存储大小和总大小，查询如下：

```
> db.movies.$_id_.dataSize();
139264
> db.movies.$_id_.storageSize();
655360
> db.movies.$_id_.totalSize();
655360
```

mongodb_data_size.txt

也可以通过集合本身获取索引数据大小：

```
> db.movies.totalIndexSize();
360448
```

mongodb_data_size.txt

storageSize 和 totalIndexSize 加起来正好是集合的 totalSize。在集合上使用 validate 方法能够得到更多与大小有关的信息。在 movies 集合上执行 validate 如下：

```
> db.movies.validate();
{
  "ns" : "mydb.movies",
  "result" : "\nvalidate\n  firstExtent:0:51a800 ns:mydb.movies\n
  lastExtent:0:558b00 ns:mydb.movies\n  # extents:4\n
  datasize?:327280 nrecords?:3883 lastExtentSize:376832\n
  padding:1\n  first extent:\n    loc:0:51a800 xnext:0:53bf00
  xprev:null\n    nsdiag:mydb.movies\n    size:5888
  firstRecord:0:51a8b0 lastRecord:0:51be90\n  3883 objects found,
  nobj:3883\n  389408 bytes data w/headers\n  327280 bytes
  data wout/headers\n  deletedList: 1100000000001000000\n
  deleted: n: 3 size: 110368\n  nIndexes:2\n
  mydb.movies.$_id_ keys:3883\n    mydb.movies.$title_1 keys:3883\n",
  "ok" : 1,
  "valid" : true,
  "lastExtentSize" : 376832
}
```

mongodb_data_size.txt

除占用空间以外，命令 validate 还提供了更多的信息，包括记录、头信息、扩展块大小（extent size）和键。

MongoDB 将数据库及其集合存储在文件系统的文件中。为了理解为存储文件分配的大小，我们来列出它们和它们的大小。在 Linux、Unix、Mac OS X 或其他任何 Unix 变种上，可以列出大小如下：

```
ls -sk1 ~/data/db
total 8549376
      0 mongod.lock
  65536 mydb.0
 131072 mydb.1
 262144 mydb.2
 524288 mydb.3
1048576 mydb.4
2096128 mydb.5
2096128 mydb.6
2096128 mydb.7
  16384 mydb.ns
 65536 test.0
131072 test.1
 16384 test.ns
```

mongodb_data_size.txt

这是从我的/data/db 目录得到的输出，你的会有所不同。不过数据库文件大小的模式应该不会有区别。这些文件对应一个数据库，每个数据库有一个命名空间文件和多个数据存储文件。不同数据库的命名空间文件大小相同：在 64 位 Snow Leopard Mac OS X 上是 16 384KB 或者 16MB。数据文件从 0 开始按顺序编号。对 mydb，模式是下面这样的。

- ❑ `mydb.0` 是 65 536KB 或者 64MB。
- ❑ `mydb.1` 是 `mydb.0` 大小的 2 倍，131 072KB 或者 128MB。
- ❑ `mydb.2`、`mydb.3`、`mydb.4`、`mydb.5` 分别是 256MB、512MB、1024MB（1GB）和 2047MB（2GB）。
- ❑ `mydb.6`、`mydb.7` 均与 `mydb.5` 大小相等，为 2GB。

MongoDB 为存储分配的块逐渐增大，默认的最大值是 2GB，到这个程度以后，每个文件就都跟最大块一样大。MongoDB 通过采用这种算法分配存储文件，最小化未使用空间和碎片。

MongoDB 还有很多细微之处，特别是在内存管理和分片上，这些都留给你来探索。本书涵盖很多产品，要介绍每一种产品的每一个细节就超出了本书的范围。

下面介绍 Membase 架构的部分基础内容。

13.2　Membase 架构

Membase 支持 Memcached 协议，所以使用了 Memcached 的应用可以很容易地将 Membase 纳入它们的应用程序栈。但 Membase 还添加了持久化和复制等 Memcached 不支持的能力。

每个 Membase 节点运行一个 `ns_server` 实例，它是节点督管者，管理着节点上的活动。客户端用 Memcached 协议或 REST 接口与 `ns_server` 交互。REST 接口是借助 Menelaus 组件实现的。Menelaus 包括一个健壮的 jQuery 层，该 jQuery 层将 REST 调用向下映射到服务器。用 Memcached 协议访问 Membase 的客户端通过 Moxi 代理访问底层数据。Moxi 作为中介在 vBuckets 的帮助下总能将客户端路由到正确的地方。要理解 vBuckets 如何正确地路由信息，需要了解 vBuckets 使用的一致性哈希。

图 13-2 演示了基于 vBuckets 路由的本质。

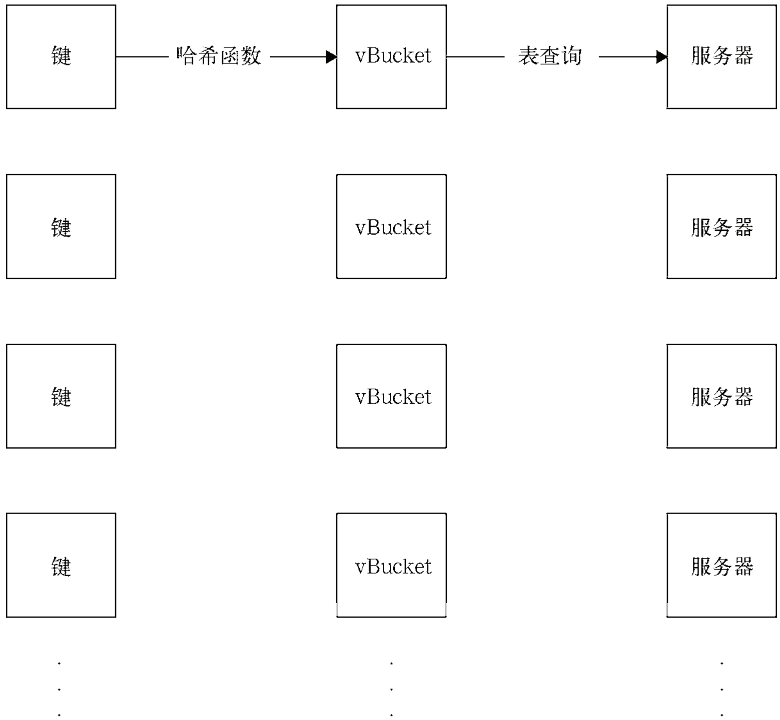


图 13-2

如图 13-2 所示，客户端请求由键所标识的数据时，请求会被映射到 vBuckets 上，然后 vBuckets 再映射到服务器上。哈希函数把键映射到 vBuckets 上，此外还支持 vBuckets 数量发生变化时重新平衡。vBuckets 自己通过查询表映射到服务器上。所以 vBuckets 到服务器的映射相对固定，重新分配 vBuckets 时物理存储不会移动。

Membase 由下列组件组成。

- ns_server：核心管理器。
- Memcached 和 Membase 引擎：Membase 构建在 Memcached 上面。网络和协议支持层都直接取自 Memcached，并包含在 Membase 中。Membase 引擎还支持一些附加功能，例如异步持久化和远程定位器数字字母协议（Telocator Alphanumeric Protocol，TAP）。
- vBucketMigrator：根据 `ns_server` 启动 `vbucketmigrator` 进程的方式，在节点间复制或传输数据。
- Moxi：Memcached 代理，支持通过 vBucket 对客户端请求路由。

Membase 还有很多内容，希望通过本节你已经理解了其中最基础的部分。更多有关 Membase 架构和性能的内容会在后面章节中陆续介绍。

13

Membase 现在是 CouchBase 的一部分，这是 Membase 和 CouchDB 合并的结果。后续的版本里，这个合并很可能会给 Membase 架构带来巨大的影响。

13.3　Hypertable 底层

Hypertable 是 HBase 的高性能替代物。Hypertable 的基本特性与 HBase 非常类似，HBase 又类似 Google Bigtable。Hypertable 其实算不上新，它和 HBase 的启动时间差不多都是 2007 年。Hypertable 运行在 HDFS 这样的分布式文件系统上。

在 HBase 里，数据以列族为中心，按行键排序存储。每个数据单元维护多个数据版本。Hypertable 支持类似的思路。Hypertable 里所有版本信息追加到行键上，版本信息由时间戳标识，每个列族每个行键所有版本的所有数据都按顺序存储。

Hypertable 提供以列族为中心的数据存储，但是其物理存储特性还受访问组（access group）的影响。Hypertable 的访问组提供一种方式，能将相关列的数据物理上存储在一起。在传统 RDBMS 中，数据按行排序，也如此存储。也就是说，两个连续行的数据通常也存储在相邻位置上。在列族存储中，物理上存储在一起的是两列数据。有了 Hypertable 访问组，就具备了将一或多列放在一个组里的灵活性。所有列都在同一个访问组里就类似传统 RDBMS，保持每列都与其他列隔离就类似面向列族数据库。

13.3.1　正则表达式支持

Hypertable 查询可以用正则表达式过滤数据，能按行键、列名和值进行匹配。Hypertable 对正则表达式的支持是用 Google 的开源正则表达式引擎 RE2 来实现的。更多有关 RE2 的细节请访问：http://code.google.com/p/re2/。RE2 是一个高速、安全和线程友好的正则表达式引擎，C++实现。RE2 为 Google 许多著名产品例如 Bigtable 和 Sawzall 提供了正则表达式支持。

RE2 语法支持大部分 Perl 兼容正则表达式（Perl Compatible Regular Expression，PCRE）、PERL，以及 VIM 支持的大部分表达式。可以在这里访问到所支持的语法列表：http://code.google.com/p/re2/wiki/Syntax。

一些 Hypertable 开发者所进行的测试（http://blog.hypertable.com/?p=103）表明 RE2 的性能大约在 `java.util.regex.Pattern` 的 3 到 50 倍之间。这些测试是在 110MB 包含 450 万唯一 ID 的数据上进行的。测试结果受数据集和数据量不同的影响会有所变化，不过结果还是能说明 RE2 高速高效的事实。

13.3.2　布隆过滤器

布隆过滤器（Bloom Filter）是一种概率数据结构，用于测试元素是否为集合成员。可以把布隆过滤器看作是 m 位的数组。空的布隆过滤器所有 m 个位置的值都是 0。现在假设有集合元素 a、b、c，用 k 个哈希函数将它们映射到布隆过滤器上，即每个元素通过 k 个不同的哈希函数映

射到布隆过滤器的 k 个位置上。元素通过哈希函数被映射到 m 位数组的某个特定位置上，则将那个位置的值设置成 1。不同的元素穿过哈希函数后，可能会被映射到布隆过滤器的同一个位置上。

假如现在要测试元素 w 是否属于集合，则要将其传递给 k 个哈希函数，输出结果映射到布隆过滤器的数组上。映射到的位置，只要任何一个值为 0，那么元素就不是集合成员。如果所有位置的值都是 1，则有两种可能，要么元素属于集合，要么只是恰好映射到因其他元素设置为 1 的位置上。所以误报是可能的，不过假阴性是不可能的。

更多有关布隆过滤器的内容（用 PERL 解释）可以访问：www.perl.com/pub/2004/04/08/bloom_filters.html。

13.4 Apache Cassandra

Apache Cassandra 非常流行，但它同时又是一种恶名昭著的 NoSQL 数据库。本书前面通过一些使用 Cassandra 的例子介绍了该存储的核心思想。本节将学习 Cassandra 的核心架构，以理解其工作机制。

13.4.1 点对点模型

大部分数据库，包括大部分知名 NoSQL 存储，都选择主–从模式来实现横向扩展和复制。即对每个集合，写提交到主节点，然后再复制到从节点。从节点增强了读而非写的可扩展性。

Cassandra 抛弃主–从模式而选择使用点对点模型。因此不存在单一主节点，所有节点都是潜在的主节点。这使得写和读都变得高度可扩展，甚至允许在部分节点失效的情况下其他节点继续提供功能。不过高可扩展性也是有代价的，在这里就是妥协强一致性。点对点模型遵从弱一致性模型。

13.4.2 基于Gossip和Anti-entropy

由于 Cassandra 采用点对点可扩展和最终一致性模型，所以建立一个协议来实现节点间通信和节点检测故障显得尤为重要。Cassandra 依靠闲话协议（gossip-based protocol）来进行节点间通信。闲话协议，“技”如其名，借用了人与人之间闲话的概念。对人来说，闲话是任意选择一个其他节点（人）来发送消息。而在 Cassandra 中，闲话更系统化，由 `Gossiper` 类定时触发。节点向 Gossiper 注册自己，这样闲话在网络中传播时节点就能收到通知。闲话主要用于大型的分布式系统，并不是特别可靠。在 Cassandra 中，`Gossiper` 类会跟踪传播闲话的节点。

按流程说，每次定时器驱动的 Gossiper 行为都要求 Gossiper 随机选择一个节点来发送消息。这个消息叫 GossipDigestSyncMessage。如果接受节点活着，就返回一个确认信息给 Gossiper。然后 Gossiper 再回应：知道了，这样就完成了一轮闲话。如果通信过程中，三步都成功了，则 Gossiper 和节点成功地共享了状态信息。如果中间有失败，则说明节点可能坏掉了。

为了检测失败，Cassandra 使用了一种算法，叫做 Phi Accrual Failure Detection。这种检测方法把节点死活之是非变成可疑程度。通过心跳检测错误的传统想法现在被连续评估可疑程度的做

13

法所取代。

闲话能保持节点同步，修复暂时性问题。更严重的问题则是通过 anti-entropy 算法发现和修复。这一过程中，列族数据通过 Merkle 树生成哈希。Merkle 树比较邻居节点的数据，如果不一致，则修复节点使其一致。在压缩操作中，Merkle 树会作为快照被创建出来。

这样就再一次确认了：Cassandra 的弱一致性可能使得它需要借助 Quorum 来避免不一致的情况发生。

13.4.3　快速写

Cassandra 的写速度极快，因为写只是追加到可用节点的提交日志，而且关键路径上也没有强加上锁。写操作分两部分，一是写入提交日志保证持久性和可恢复性，二来更新内存数据结构。只有在写入提交日志成功后才写内存数据结构。机器上通常会专门为提交日志准备磁盘，因为所有对提交日志的写都是顺序的，这样能最大化磁盘吞吐。内存数据结构有一个阈值，是基于数据大小和对象数量计算的，如果内存数据超过了阈值，它就会把自己转储到磁盘上。

所有磁盘写都是顺序的，同时也会生成索引以便基于行键高效查询。这些索引跟数据一起持久化。随着时间推移，磁盘上多出很多这样的日志，后台运行的合并进程会将不同日志整理成一个。这种压缩过程在 SSTable 里合并数据，SSTable 是底层存储格式。压缩过程还会执行键合理化、列合并、删除打上删除标记的数据和创建新索引。

13.4.4　提示移交

写操作执行过程中，如果 Cassandra 节点不可用，那么发送到该节点的请求就可能失败。如果节点被从网络中剥离开来，写操作也可能无法重新处理。为了处理这些情况，Cassandra 引入了提示移交（hinted handoff）的概念。这个概念最好通过例子来说明，所以我们假设网络中有两个节点 *X* 和 *Y*。现在尝试在 *X* 上执行写操作，但是 *X* 不在线，于是将写操作发送给 *Y*。*Y* 存储信息时附带提示说，这个写本来是给 *X* 的，什么时候 *X* 上线了，请把数据发给它。

Basho Riak（www.basho.com/products_riak_overview.php）是另一个受 Amazon Dynamo 启发的数据库，它也利用这个概念协调写操作。

除了这些有趣的，经常被谈论到的 Cassandra 的特性和底层结构外，我还需要提一点，即 Cassandra 建立在阶段事件驱动架构（Staged Event-Driven Architecture，SEDA）上。更多有关 SEDA 的内容请参阅：www.eecs.harvard.edu/~mdw/proj/seda/。

下一个要介绍的产品是非常优秀的键/值存储 Berkeley DB。Berkeley DB 是众多 NoSQL 产品的底层存储，而 Berkeley DB 本身也可用作 NoSQL 产品。

13.5　Berkeley DB

Berkeley DB 有下列三个截然不同的版本，支持多个配置。

- Berkeley DB：C 实现的键/值存储，是最原汁原味的版本。

- Berkeley DB Java 版（Java 标准版）：Java 写的键/值存储，可以很容易地集成到 Java 栈里。
- Berkeley DB XML：C++编写，这个版本封装了键/值存储，使其行为更像带索引的优化的 XML 存储系统。

Berkeley DB 又称 BDB，骨子里是键/值存储。虽然核心非常简单，但是 BDB 支持许多不同配置。比如可以配置成支持并发非阻塞访问，或者支持事务。它也可以横向扩展，成为高可用的主-从副本集群。

BDB 是一种键/值存储，它很纯粹，不对键/值对的结构作任何假设，因此很容易在 BDB 上建立高层 API、查询和建模抽象，这样有利于快速有效地存储特定应用数据，又没有将其转换成抽象数据格式的开销。这样简单优雅的设计所带来的灵活性让 BDB 既可以存储结构化数据，又可以存储半结构化数据。

BDB 可以内存存储方式运行，保存小量数据，也可以配置为大型数据存储，带一个内存缓存。BDB 包含叫做环境（enviornment）的高层抽象，借助它，一个物理安装中可以建立多个数据库。一个环境可以有多个数据库。你需要先打开环境，再打开数据库，才能向其中写入数据或从中读出数据。交互完成后应关闭数据库和环境以优化资源使用。数据库的每个条目都是一个键/值对。键通常是唯一的，不过也可以有重复。值通过键访问。获取的值可以更新并存回数据库。多个值可以用游标遍历。游标支持遍历集合，每次一个地处理集合元素。另外 BDB 也支持事务和并发访问。

键/值对里的键几乎总是作主键用，主键有索引。值的其他属性可以用作次级索引，次级索引维护在单独的数据库里。因此存储着键/值对的数据库有时也称为主数据库。

BDB 运行时是进程内数据存储，所以使用 C、C++、C#、Java 或者脚本语言 API 从对应的程序里访问 BDB 时，要静态或者动态链接之。

存储配置

键/值对可以被存储成四种数据结构：B 树、哈希、队列和 Recno。

1. B 树存储

B 树没什么可介绍的，如果需要复习它的定义，可以阅读免费提供的在线文档：www.bluerwhite.org/btree/。它是一种平衡树结构，保持元素有序性的同时还支持快速顺序访问、插入和删除。键和值可以是任意数据类型。BDB 里 B 树还支持对重复值的访问。如果需要用复杂数据类型作为键，那 B 树存储是很好的选择。如果访问模式是访问连续或相邻记录，它也是非常棒的选择。B 树保存大量元数据以实现很高的执行效率。绝大部分 BDB 应用会选择 B 树存储。

2. 哈希存储

与 B 树类似，哈希也支持复杂类型做键。与 B 树相比，哈希的结构更线性化。BDB 的哈希结构也支持重复值。

尽管 B 树和哈希都支持复杂键，但在数据集远大于可用内存时哈希通常优于 B 树，因为 B 树保存更多的元数据，内存缓存中可能放不下 B 树的元数据。极端情况下需要频繁地通过文件来获取 B 树的元数据和实际数据，导致每次操作触及多个 I/O。哈希的访问方法能最小化访问数据

13

记录所需的 I/O，因此在这种极端情况下，可能比 B 树性能好一些。

3. 队列存储

队列是一系列顺序存储的定长记录，键限制为整数类型的逻辑记录号。顺序追加记录可使得写速度飞快。如果你对 Apache Cassandra 追加日志的写速度印象深刻，那就给 BDB 队列访问方法一个机会，你绝不会失望。这种配置还支持从队列头部高效地读取和更新。队列还支持行级锁，即便是在并发处理情况下也能保证事务完整性。

4. Recno 存储

Recno 与队列类似但支持变长记录，和队列相同的是 Recno 键也限定为整数。

综上，不同配置允许在集合中存储任意数据类型。除了你自己的模型外，没有任何其他固定的结构。极端情况是一个集合的两个值可以存储完全不同的类型。值可以是复杂的类型，这些类型可以用来表示 JSON 文档、复杂数据结构或者结构化的数据集。实际上的唯一限制是值必须能被序列化为字节数组。单个键或值可以达到 4GB 大小。

次级索引允许在值属性上过滤。主数据库并不以表格形式存储数据，所以对稀疏数据集而言，不存在的属性也不会存储。如果缺乏创建索引用的属性字段，次级索引就会跳过所有记录。总的来说，存储是紧凑高效的。

13.6　小结

本章只介绍了一些数据库的部分内部细节，同样的思路未必能准确适用于其他存储。对架构和内部结构的了解可以在多个不同层次上进行，从概念综览到深入代码。本章主要限定在概念层次上，以便于所有人阅读。不过本章应该赋予了你开始自己探索的工具和知识。

Part 4

第四部分

掌握 NoSQL

本部分内容

第 14 章 选择 NoSQL

本章内容

- 了解 NoSQL 产品的优劣势
- 比较 NoSQL 产品
- 根据性能基准评估 NoSQL 产品

不是所有 NoSQL 数据库都相似，它们被造出来也不是为了解决同样的问题，因此企图通过比较从中选出最好的很可能会徒劳无功。不过，了解什么数据库更适合哪种情景和上下文非常重要。本章介绍的事实和意见能帮助你比较各种 NoSQL 产品。我们将根据功能、性能和基于上下文的标准对 NoSQL 数据库进行分类和权衡。

NoSQL 的演进可以和这些年编程语言的快速发展相提并论。多种编程语言让我们能用正确的语言处理正确的任务，因而开发者的简历上经常出现多种语言。开发者使用多种编程语言常比作一个人通晓多国语言，或语言通。语言通能有效地避免缺少某种语言知识带来的障碍。类似地，采纳多种编程语言称为*多语言编程*。多语言编程常被看作是聪明的编程方式，因为可以使用适当的语言处理手头的任务。同样，很明显一个数据库无法满足各种需求，采用多种数据库则是明智的策略。知道和使用多种数据库产品的人可以算是数据库的*语言通*。

NoSQL 数据库有许多不同形式，逻辑分组的第一种方式是按功能。通常来说，问题的解决方案很容易映射到所需的功能上。

14.1 比较 NoSQL 产品

本节基于下列功能特性比较各种 NoSQL：

- 可扩展性
- 事务完整性和一致性
- 数据建模
- 查询支持
- 访问和接口可用性

14.1.1 可扩展性

虽然所有 NoSQL 数据库都承诺横向可扩展，但它们应对挑战的水平却不尽相同。Bigtable

的相似产品 HBase 和 Hypertable 暂时处于领先位置，内存存储（例如 Membase 或 Redis）和文档数据库（例如 MongoDB 和 Couchcbase Server）紧随其后。它们之间的差异随着数据量的增大而放大，特别是在达到 PB 量级以后。

前面几章里，我们深入了解了大部分主流 NoSQL 数据库的存储架构。Bigtable 类产品推动了大记录和大集合的存储。Bigtable 模型支持存储大量的列和海量的行记录。数据可以是稀疏的，即很多列没有数据。当然 Bigtable 模型不会浪费空间，没有值的单元格不会被存储。

> HBase 集群中列和行的数量理论上没有限制。列族的数量被限定为大约 100 个。行的数量可以不断增加，只要有新节点保存数据就行。列的个数很少超过几百个。处理数据时，太多的列会带来逻辑上的挑战。

Google 引领了以列族为中心的数据存储的变革，并用它存储爬虫带回家的不断增长的网络索引。过去几年里 Web 在无限制地成长，Google 需要一种能随索引增长而增长的存储，由此诞生了 Bigtable 和其他相似产品，它们天生就支持横向扩展，只受限于集群硬件数量，因而不能衍生出集群中新的节点。过去几年里，Google Bigtable 被成功地用于存储和访问各种不同类型的海量数据。

HBase wiki 的 Powered By 页面（http://wiki.apache.org/hadoop/Hbase/PoweredBy）罗列了许多 HBase 用户。这些用户清楚地说明了 HBase 的扩展能力。

> 下面段落中将说明 HBase 的能力，而同时 Hypertable（另一个类 Google Bigtable 产品）也提供了同样的能力。

Meetup（www.meetup.com）是一个很流行的网站，方便了用户组和兴趣组来组织当地的活动及会议。Meetup 2001 年还是一个不知名的小站，现在已经增长到在 100 个国家拥有 800 万成员、6.5 万余名组织者、8 万多 meetup 组和每周 5 万场会议（http://online.wsj.com/article/SB10001424052748704170404575624733792905708.html）。Meetup 是 HBase 的用户。所有小组活动都直接写入 HBase 并按成员索引，成员的自定义 feed 是直接从 HBase 里提供的。

Facebook 是另一个 HBase 大用户。Facebook Message 建在 HBase 上。2010 年 Facebook 是互联网的头号目标网站。它已经发展了超过 5 亿活跃用户（www.facebook.com/press/info.php?statistics），从用户数量来看算是世界上最大的应用。Facebook Message 将聊天、短信和邮件集成起来。每个月有数千亿消息通过这个消息基础设施发送。Facebook 工程师团队分享了几篇他们在 Facebook Message 基础设施上使用 HBase 的笔记，地址：www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919。

在扩展方面 HBase 天生就具备一些优势。HBase 支持自动负载平衡、故障转移、压缩和单服务器多分片。HBase 和 HDFS（Hadoop 分布式文件系统）配合得很好。从前面章节我们知道，HDFS 能通过复制和自动平衡轻松容纳跨越多服务器的大文件。Facebook 选择 HBase 来利用这些

14

功能特性。从他们服务的消息和用户的数量来看，HBase 是必需的。Facebook 工程师在笔记中还提到基础设施中的消息具备简短、易变、临时的特征，而且旧消息很少再访问。对于即时查询并不重要的场景来说，HBase，或者说一般的类 Bigtable 产品特别适合。HBase 支持查询，但是说到查询的性能，它并不能作为 RDBMS 很好的替代品。而像 Google 应用服务引擎（GAE）这样的基础设施却能在 Bigtable 上成功地开放出数据建模 API，并支持高阶查询。更多有关查询的信息会在后面 14.1.4 节介绍。

所以如果需要极端扩展性的话，列族 NoSQL 数据库似乎是很好的选择。但是这类数据库可能并非所有系统的最佳选择，特别是涉及实时事务处理的情况。如果事务完整性非常重要，RDBMS 和 NoSQL 相比常常是更好的选择。如果能接受更弱的一致性，那最终一致性 NoSQL 存储，比如 Cassandra 或 Riak 也是值得考虑的。Amazon 已经证明了大规模电子商务也许能用最终一致性数据存储，不过 Amazon 之外就很难再找到此模型适用的例子了。像 Cassandra 这样的数据库遵循了 Amazon Dynamo 的范式，支持最终一致性。Cassandra 承诺了快得不可思议的读写速度。Cassandra 也支持类 Bigtable 的列族数据模型。Amazon Dynamo 还启发了 Riak。除了是最终一致性存储外，Riak 还支持文档存储。Cassandra 和 Riak 的横向扩展能力都不错，不过要是可扩展性至关重要，那我还是会放弃最终一致性存储，转投 HBase 或者 Hypertable 的票。写吞吐和延迟更为重要的场景中，可能最终一致性存储比列族按序存储更好。因此如果需要横向可扩展性和较高的写吞吐，可以考虑 Cassandra 或 Riak。不过这种情况也可以考虑混合策略，逻辑上将写和读、分析分开来，两个任务各用不同的数据库。

如果大量数据会以惊人的快节奏出现，例如交易数据或广告点击跟踪数据，那么单靠列族存储无法提供完整的解决方案。在这些存储里保存大量增长的数据，用 MapReduce 进行批量查询和数据挖掘是比较审慎的做法，不过你可能还需要更轻快的东西，既支持快速写，又支持实时处理。没有什么比在内存里处理数据更快的了，有些 NoSQL 能将数据保存在内存里，可用容量填满后再刷到磁盘上，利用这种 NoSQL 是很好的选择。MongoDB 和 Redis 都遵循这一策略。目前 MongoDB 使用 `mmap`，而 Redis 自定义从内存到磁盘的映射。MongoDB 和 Redis 都在积极调整内存映射特性，它们还在继续发展。如果需要实时数据处理和用于数据挖掘的可扩展存储，选择 MongoDB 或 Redis 搭配 HBase 或 Hypertable 不错。Memcached 和 Membase 可以用来替换 MongoDB 或 Redis。Memcached 和 Membase 作为高速缓存放在列族存储前面，对它是很好的补充。实际上，为这种场景在基于 Hadoop 的系统上高效使用 Membase 也有先例。随着 Membase 和 CouchDB 合并，很可能会出现同时支持以高速缓存为中心和以分布式可扩展存储为中心的功能特性的 NoSQL 产品。

如果你的数据需求达到了 Google 或者 Facebook 的级别，可扩展性就非常重要，尽管如此，并不是所有应用都能长到那么大。对那些比这类扩展性系统小得多的应用来说，可伸缩系统可能更具相关性，但是有时企图让软件可扩展会成为过度工程。一定要避免不必要的复杂性。

在许多系统里，数据完整性和事务性一致比其他需求都更重要。NoSQL 能够用于这类系统吗？

14.1.2 事务完整性和一致性

事务完整性只和数据创建、修改、更新及删除有关。因此事务完整性相关问题和纯数据仓库及数据挖掘无关。这也意味着以批处理为中心的、基于 Hadoop 的数据分析与事务性需求无关。

许多数据，比如网络流量日志、社交网络状态更新（包括微博）、广告点击、道路交通数据、交易数据和游戏分数主要（如果不是完全）是一次写多次读。像这样写一次读多次的数据对事务的需求有限，甚至没有。

有些数据虽然已更新和删除，但是修改通常只限于单记录而非数据集的某个范围。有时更新非常频繁且涉及范围操作。如果范围操作很常见而且需要保持更新的一致性，那 RDBMS 才是最佳选择。如果单个条目的原子性已然足够，那么列族数据库、文档数据库和部分键/值存储都可以。如果需要事务完整性但是可以容纳暂时的窗口不一致，最终一致性也是一种选择。

> NoSQL 的对手们认为非关系型健壮数据库的主要弱点是缺少 ACID 支持。不过，许多数据只需要很少甚至无需事务性支持。这样的数据就能受益于 NoSQL 产品可扩展的、优雅的架构。NoSQL 数据库使用 MapReduce 进行大规模并行处理的能力能够帮助有效地处理和挖掘大数据。切勿为不必要的事务完整性而烦恼。

HBase 和 Hypertable 提供行级的原子更新及一致性状态（在 Paxos 的帮助下）。MongoDB 提供文档级别的原子更新。所有主–从复制模式的 NoSQL 数据库都隐式支持事务完整性。

14.1.3 数据模型

RDBMS 的数据建模方法非常一致。数据模型底层是关系代数，理论完备且已标准化。因此数据建模和规范化的方法能够被很好地吸收理解并形成文档。在 NoSQL 世界里没有这样标准化的完备的数据模型，这是因为各种 NoSQL 产品的架构和它们想解决的问题并不相同。

如果需要 RDBMS 的数据模型进行存储和查询，而且任何情况下都无法跳过这些定义，那就别用 NoSQL。如果喜欢 SQL 类型查询又能接受非关系型存储模型，还是有一些 NoSQL 可以选择的。

像 MongoDB 这样的文档数据库提供了从 RDBMS 逐步过渡的文档模型的路径。MongoDB 支持类 SQL 查询、基本的关系型引用和数据库对象，这些对象的设计从标准的基于数据表、列的模型中汲取了大量灵感。如果用 NoSQL 的主要原因是可以使用宽松的数据结构，那么 MongoDB 肯定是开始使用 NoSQL 的绝佳选择。

很多以 Web 为中心的业务中都使用了 MongoDB。Foursquare 可能是最知名的 MongoDB 用户了。此外 MongoDB 的用户还包括了 Shutterfly、bit.ly、etsy 和 sourceforge。它们中间的许多之所以选择 MongoDB，正是因为它支持灵活的数据模型，能提供快速的读写能力。Web 应用发展通常很快，开发人员不断修改底层 RDBMS 模型常常会减慢应用发展的速度，特别是在修改频繁，

有时改动又很大时。此外数据迁移也为修改数据定义带来了很多挑战。

MongoDB 对 Web 框架集成的支持很好。作为最流行的 Web 应用程序框架之一，Rails 就可以有效地集成 MongoDB。Rails 应用中的数据通过对象映射器持久化。因此可以用 MongoDB 替换掉 RDBMS。更多有关 Rails3 集成的内容请访问：www.mongodb.org/display/DOCS/Rails+3+-+Getting+Started。

对 Java Web 程序员来说，Spring 的 Spring Data 项目对 MongoDB 提供了非常好的支持。更多有关 Spring Data 对 MongoDB 支持的内容请访问：www.springsource.org/node/3032。事实上不只是 MongoDB，Spring Data 项目还支持了很多其他 NoSQL 产品，包括 Redis、Riak、CouchDB、Neo4j 和 Hadoop。要了解更多细节请访问 Spring Data 项目主页：www.springsource.org/spring-data。

MongoDB 把数据保存在内存中，需要时再写入磁盘，很像持久化缓存。所以 MongoDB 也可以被看作是介于 RDBMS 和内存存储（或简单文件结构）之间的选择。很多 Web 应用例如实时分析、评论系统、评分存储系统、内容管理软件、用户数据系统，甚至是日志应用都受益于 MongoDB 提供的易于修改的数据结构定义。不仅如此，MongoDB 的类 RDBMS 查询能力、将数据分离到类似于表的不同集合中的能力也是这些应用喜欢的特性。

Apache CouchDB 是另一个可以替代 MongoDB 的文档数据库。CouchDB 的主要开发者将他们的公司 CouchOne 与 Membase 合并，所以 Apache CouchDB 现在改名为 Couchbase server。Couchbase 以 Couchbase Server 的形式提供 Apache CouchDB 和 GeoCouch 的打包版本，同时予以技术支持。

Couchbase 是遵从 Web 标准的典范。Couchbase 主要通过 RESTful 的 HTTP 交互来访问，而且它在以 Web 为中心的路上比任何其他数据库都走得更远。Couchbase 中包括一个 Web 服务器，它用 Erlang OTP 构建。这意味着实际上可以用 Couchbase 来开发整个应用。借助 Membase 管理速度和吞吐量的能力，未来版本中 Couchbase 还会增加通过 Memcached 协议访问的接口。Couchbase 还计划向上扩展，利用 Membase 的弹性能力无缝扩展到多个节点上。Couchbase 非常强大，功能也很丰富，但是它占用的空间却很小。轻巧的特质使得它非常适于安装在智能手机或者嵌入式设备中。更多有关移动 Couchbase 的内容请访问：www.couchbase.com/products-and-services/mobile-couchbase。

Couchbase 模型支持 REST 风格的数据管理。CouchDB 数据库可以包含 JSON 文档，同时还可以附带元数据或支持文件作为附件。所有在数据上执行的操作（创建、获取、更新和删除）都通过 RESTful 的 HTTP 请求来完成。Couchbase 集群上长时间运行的复杂查询则利用 MapReduce。

> REST，即表示性状态转移，代表一种适用于分布式多媒体系统（比如万维网）的软件架构风格。术语 REST 是由 Roy Fielding 在其博士论文中首次引入和定义的。阅读更多有关 REST 的内容请访问：www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm。

1. 不只是映射表

内存数据库和缓存里，最出名的数据结构当属映射表/哈希表。映射表保存键/值对，支持快速访问数据。NoSQL 内存数据库用文件系统持久化内存里的数据，这样系统重启后，数据不会丢失。除了映射表，许多 NoSQL 内存数据库还支持其他数据结构，这样一来，在缓存之外它们也有了大展身手的机会。

Berkeley DB 最基本的功能是保存二进制键/值对。其存储引擎不会在键/值对上附加任何元数据。对象这样的高层抽象，主要通过上层持久化 API 或对象包装器保存到 Berkeley DB 中。

Membase 支持 Memcached 文本和二进制协议、键/值对存储、副本管理和一致性哈希，可以在不影响客户端的前提下增加或减少集群的服务器数量。Redis 的特点则稍有不同，它支持大部分流行数据结构开箱即用，号称"数据结构"服务器。除映射表外，Redis 还支持列表、集合、有序集和字符串。此外 Redis 甚至具备几近事务的能力，可以将多个相互独立的操作看成一个原子操作。

如果需要带文件存储的内存数据库，为了作出最合适的选择，应考虑产品支持的数据模型。很多时候键/值存储足矣，如果需求列表里还包含这些字眼：强大、稳定、分布式，能支持大量用户和活动负载，那选 Membase 基本错不了。

2. HBase 和 Hypertable

前面谈到可扩展性时，我举双手双脚支持列族数据库。但是要说到支持丰富的数据模型，这些断然不是最佳的选择。行键查找和列族为中心的模型一般来说还是不够用。不过有了抽象层的帮忙，一切又有了可能。

Google 不仅掀起了列族数据库的革命，还为自己的列族数据库增加了数据建模抽象层。在 GAE 中，可以通过 Python 和 Java 实现丰富的数据建模（第 10 章介绍了这个主题）。有了 DataNucleus 对 JDO 和 JPA 的支持，就能在 HBase 和 Hypertable 上使用流行的 Java 对象模型，将数据持久化到 HBase 和 Hypertable。除此以外，Django 对应用程序引擎的非关系型支持也是很好的参考。

14.1.4 查询支持

如果把挑选 NoSQL 数据库比作拼图，存储是一块，那查询就是另一大块。要做出审慎的选择，高效易用的查询不可或缺。这一点在创建与人交互的应用时尤其重要。SQL 为 RDBMS 带来了繁荣，因为它让数据的访问和查询变得容易。语法语义的标准化，让它显得格外诱人。本书第 1 章谈到了 NoSQL 一众需要类 SQL 查询语言，后续章节描述了各自的实现。

文档数据库里，MongoDB 的查询能力可谓鹤立鸡群。要说"好"，其实是相对的，不同的人对什么更好有不同的看法。我看重三点：与 SQL 相似、语法简单、容易学习。话说如果掌握了 CouchDB 视图和设计文档的概念，CouchDB 的查询能力其实一样强大，而且更直观。但是 CouchDB 视图是一个新概念，前期会给开发者带来一定的挑战。

基于内存的键/值存储里，没有谁能比 Redis 的查询能力更丰富。它的查询方法算得上最全面。命令文档之美，更是锦上添花。有关 Redis 查询方法的内容请参阅：http://redis.io/commands。

像 HBase 这样的列族存储几乎没有提供多少查询能力。不过在 Hive 项目的帮助下，用类 SQL 的语法查询 HBase 已经得到实现。第 12 章会介绍 Hive。Hypertable 的查询语言是 HQL，它同时也支持 Hive。

有了 Hive，人们自然就会问：是用它进行日常操作数据，还是批量处理或者商业智能计算？相比 SQL 之与 RDBMS，Hive 交互性没有那么好。Hive 形似 SQL，实际上是对 MapReduce 风格的数据处理的一种抽象。它支持用类 SQL 语法代替 map、reduce 函数来进行批量数据处理操作。

14.1.5　接口可用性

MongoDB 有驱动的概念，用来访问 MongoDB 各种主流类库的驱动都有。CouchDB 用标准的 Web 方式访问，所以任何支持 Web 通讯的语言都能访问 CouchDB。一些语言封装的 CouchDB 类库运作起来很像 MongoDB 驱动，只不过 CouchDB 总有 RESTful HTTP 接口罢了。

Redis、Membase、Riak、HBase、Hypertable、Cassandra 和 Voldemort 都支持大部分主流语言。其中许多都是用像 Thrift 这样的语言中立的服务层来封装，或者底层用到了像 Avro 这样的序列化机制。所以理解各种序列格式的性能特征就变得很重要。

对 JVM 上的序列化格式，jvm-serializers 项目很好地分析它们的性能特点，地址是 https://github.com/eishay/jvmserializers/wiki/。这个项目测量了许多种数据格式的性能，覆盖到的格式如下。

- protobuf 2.3.0。Google 数据交换格式。http://code.google.com/p/protobuf/
- thrift 0.4.0。Facebook 开源，HBase、Hypertable 和 Cassandra 等 NoSQL 产品使用。http://incubator.apache.org/thrift/
- avro 1.3.3。Apache 项目。替代一些 NoSQL 产品中的 Thrift。http://avro.apache.org/
- kryo 1.03。Java 对象图序列化框架。http://code.google.com/p/kryo/
- hessian 4.0.3。二进制 Web 服务协议。http://hessian.caucho.com/
- sbinary 0.3.1-SNAPSHOT。描述 Scala 类型的二进制格式。https://github.com/harrah/sbinary
- google-gson 1.6。Java 对象转 JSON 的类库。http://code.google.com/p/google-gson/
- jackson 1.7.1。Java 的 JSON 处理器。http://jackson.codehaus.org/
- javolution 5.5.1。实时和嵌入系统。Java。http://javolution.org/
- protostuff 1.0.0.M7。利用 protobuf 进行序列化。http://code.google.com/p/protostuff/
- woodstox 4.0.7。高性能 XML 处理器。http://woodstox.codehaus.org/
- aalto 0.9.5。Aalto XML 处理器。www.cowtowncoder.com/hatchery/aalto/index.html
- fast-infoset 1.2.6。Fast infoset 二进制 XML 的开源实现。http://fi.java.net/
- xstream 1.3.1。XML 序列化类库及备份。http://xstream.codehaus.org/

性能数据是在 JVM 上运行得到的，不过结果可能也跟其他平台相关。结果显示 protobuf、protostuff、kryo 和手动处理序列化和反序列化的性能最好。如果看序列化大小及压缩后的大小，那 Kryo 和 Avro 格式最高效。

在查看了格式性能之后，下一节我们将继续探究 NoSQL 产品自身的性能测试。

14.2 性能测试

YCSB（Yahoo! Cloud Services Benchmark，Yahoo!云性能测试服务）是最有名的 NoSQL 性能比较工具。虽然它有一定的局限性，但提供的 NoSQL 产品分析还是比较全面。YCSB 包含两个重要的工具：

- 负载生成器（workload generator）
- 负载生成器使用的样例负载

YCSB 地址是：https://github.com/brianfrankcooper/YCSB。Yahoo!的性能测试中包含了许多种 NoSQL 产品。最新公布的结果中包含下列产品：

- Sherpa/PNUT 类 Bigtable 系统（HBase、Hypertable、HTable、Megastore）
- Azure
- Apache Cassandra
- Amazon Web Services（S3、SimpleDB、EBS）
- CouchDB
- Voldemort
- Dynomite
- Tokyo Cabinet
- Redis
- MongoDB

测试分层执行，每层都测量延迟和吞吐。第 1 层关注给定硬件下最大负载时的性能。硬件不变，负载持续增加直至硬件饱和为止。第 2 层关注可扩展性。硬件数量随负载增加而增加。第 2 层测量负载和硬件等比增加时的系统延迟。

负载有各种不同的配置，用来测试在平衡和满负荷等不同条件下的性能和可扩展性。下面介绍测试用例。

14.2.1 50/50 的读和更新

50/50 的读和更新是读少写多的测试用例。结果显示这种情况下 Apache Ccassandra 的读和更新延迟要优于其他竞争对手，HBase 紧随其后。Cassandra 每秒能执行超过 1 万次操作（50/50 读和更新），平均读延迟 25 毫秒。更新的性能比读好，同样的负载下平均延迟只有 10 毫秒，每秒能执行超过 1 万次操作。YCSB 里除了 NoSQL 还有 MySQL。虽然这一章里我忽略了 RDBMS 和 NoSQL 的性能对比，不过我发现很有趣的是，MySQL 读和更新的延迟只在每秒执行 4000 个左右的操作时才有可比性，每秒执行操作数超过 5000 以后延迟就迅速攀升。

14.2.2 95/5 的读和更新

95/5 的读和更新是读多写少的测试用例。这个用例证明了本书描述过的一些理论，比如对有序列族存储来说，连续范围的读取性能最好。HBase 的读性能表现很稳定，与每秒操作数无关。

5%的更新在 HBase 里几乎没有延迟。只读情况下 MySQL 的性能最好，这可能跟缓存有关。如果结合分布式缓存 Memcached 或者 Membase，HBase 的读取性能可与 MySQL 比肩，而且负载增加时扩展效果还会更好。Cassandra 在读多写少的用例中表现出十分出色的性能，超过了 HBase。不过别忘了 Cassandra 是最终一致性模型，而且所有写都要追加到 commit log 中。

14.2.3　扫描

HBase 的扫描性能注定要超过其他数据库，包括 1 到 100 条记录和范围扫描的情况，测试证明了这一点。Cassandra 的扫描性能则无法预测。

14.2.4　可扩展性测试

Cassandra 和 HBase 在负载增加、硬件增加的情况下性能表现相对稳定。一些结果显示 HBase 节点少于三个时比较不稳定。添加硬件非常重要的一方面是弹性。弹性衡量添加新节点后对数据的重新均衡（rebalance）。Cassandra 这方面表现较差，看起来需要很长时间才能稳定下来。HBase 每次重新平衡都表现出一致的性能，这跟压缩有关。

一如早先提到过的，性能测试只是一方面，如果纯靠测试结果作决策，很可能会被误导。此外产品不断更新，同一个产品不同版本测试的结果也不一样。综合考虑性能指标和功能比较，而非单纯依靠某方面，才是更明智的做法。

> Hypertable 测试不是 YCSB 测试中的部分，二者相互独立。YCSB 测试更宽泛，涉及一系列 NoSQL 和 RDBMS 产品，Hypertable 测试更专注于考察有序列族存储的性能。

14.2.5　Hypertable测试

Hypertable 团队提供了一套测试来比较 HBase 和 Hypertable 这两大 Google Bigtable 类产品。测试结果很有意思，执行的测试与 Google Bigtable 论文提出的一致。可以在线阅读 Bigtable 研究论文第 7 节来理解测试的内容，地址是：http://labs.google.com/papers/bigtable.html。

结果一致表明在大多数情况下 Hypertable 的性能都比 HBase 好。测试及其结果的细节可以访问：www.hypertable.com/pub/perfeval/test1/。下面解释其中一些重要的发现。

Hypertable 根据负载动态调整分配给子系统的内存。读多时，Hypertable 把大部分内存分配给缓存。HBase 缓存分配是固定的，即 Java 堆的 20%。从延迟的角度看，很明显 Hypertable 的延迟总比 HBase 小，数据越小，差异越鲜明。数据只有 2GB 以下时，所有数据都能加载到缓存里。

随机写、顺序读和扫描的测试结果表明在这些用例中 Hypertable 的性能都比 HBase 好。如果用集群管理大量数据，有时性能差别会影响成本。更好的性能即更低的计算周期和资源消耗，意味着能节约更多的成本。

不同的产品提供商提供了许多测试，包括下面这些：

- Tokyo Cabinet Benchmark。http://tokyocabinet.sourceforge.net/benchmark.pdf
- How fast is Redis。http://redis.io/topics/benchmarks
- Riak benchmark。https://bitbucket.org/basho/basho_bench/
- VoltDB: Key/value benchmarking。http://voltdb.com/blog/key/value-benchmarking
- Sort benchmark。http://sortbenchmark.org/

14.3 背景比较

前两节按功能和性能比较了不同的 NoSQL。本节介绍一些 NoSQL 产品的背景相关信息，及引发这些产品的创建和发展的条件。

各种 NoSQL 产品并不相同。NoSQL 产品的功能和性能也不全一样。每个 NoSQL 产品都有它自己的历史、动机、场景和独特的价值主张。设身处地从这些观点，特别是产品的历史与演变的角度去看，有助于更好地理解哪种 NoSQL 产品更适合完成手头的任务。

对文档数据库，可以浏览下面的在线资源。

- CouchDB。Erlang Factory 2009 会议的一个视频（www.erlang-factory.com/conference/SFBayAreaErlangFactory2009/speakers/DamienKatz）中，CouchDB 创始人 Damien Katz 从个人角度谈论了 CouchDB 开发的历史，他谈到激发他创造 CouchDB 的灵感，以及为什么决定让妻子和孩子们搬到一个更便宜的地方，节省支出来开发数据库。他还谈到了切换到 Erlang 的决定以及加入 Apache 基金会的转变。这个视频揭示了产品背后的动机和因由。
- MongoDB。推荐阅读 Kristina Chodrow 在她的博客上写的非官方历史：www.snailinaturtleneck.com/blog/2010/08/23/history-of-mongodb/。

对键/值数据库，可以浏览的资源如下。

- Redis。阅读 Antirez (Salavtore Sanfi llippo)决定将 lloogg.com 从 MySQL 切换成 Redis 以后写的邮件列表（http://groups.google.com/group/redis-db/browse_thread/thread/0c706a43bc78b0e5/17c21c48642e4936?#17c21c48642e4936）。
- Tokyo Cabinet。阅读产品首页上写的 Tokyo Cabinet 的价值主张，地址为：http://fallabs.com/tokyocabinet/。
- Kyoto Cabinet。开发 Tokyo Cabinet 的那伙人创建了一个新产品 Kyoto Cabinet。细节请参阅：http://fallabs.com/kyotocabinet/。

Bigtable 和 Dynamo 类产品 HBase、Hypertable、Cassandra 和 Riak 的历史主要是想要复制 Google 和 Amazon 的成功。学习它们的历史不外乎是复制 Google 和 Amazon 的好点子，几乎没有其他内容。当然复制点子也不是那么容易，需要有一个探索和创新的过程。随着用户把这些产品投入到新的、不同的场景中，产品会快速地发展。它们在发展过程中很可能会引入许多超越原有实现的新特性。

NoSQL 是新兴领域，虽然理解产品发展的背景很有价值，但许多 NoSQL 产品的历史还尚在书写中。

14.4　小结

本章提供了流行 NoSQL 产品的比较，虽然简洁，但也蛮有意义的。选择 NoSQL 产品需要细心，而且只有在了解了产品的功能、性能特征和历史以后才能决定。

本章并未解释所有特性，也没有提供模型来帮助你选择产品，而是建立以之前的章节为基础，着重演示一些重要的事实，总结一些基本的观点。无论如何，选择权还是在你自己手中。

第 15 章

共　　存

本章内容

- ❑ 为多种持久化技术做准备
- ❑ 了解适用于静态数据的数据库技术
- ❑ 选择适合的数据库来简化应用开发
- ❑ 组合使用 RDBMS 和 NoSQL 产品

NoSQL 的出现和它愈演愈烈的趋势让开发者们开始思考：NoSQL 是否能代替 RDBMS？有说法认为 RDBMS 已死，而 NoSQL 将主导下一代数据库技术。另外一些言论试图证明 NoSQL 不过是昙花一现。这两种言论都过于偏激。NoSQL 和 RDBMS 都很重要，各有它们的用武之地，这两种技术和平共处更为现实。技术多元化是常态，共存不仅仅是今天，也是未来的趋向。为了共存，需要做哪些准备？有什么样的流程？本章将着重介绍这些内容。

本章首先介绍在热门的开源 RDBMS 产品——MySQL 中如何利用 NoSQL 思想。接下来介绍在数据仓库和商业智能领域里，不可变的数据对数据库的要求。除此之外，还会介绍在某些情况下，恰当地选择数据库技术，能使得应用开发变得更轻松。

15.1　MySQL 用作 NoSQL

到目前为止，本书一直都把 RDBMS 和 NoSQL 看作两种全然不同的技术。在理解与 SQL 和关系表对立的 NoSQL 时，这种观点非常重要。但是这两类思想并非完全割裂，它们也有着很多相同的理念。一个例子就是 RDBMS 和一些 NoSQL 的索引常用的 B 树或类 B 树结构。不过即便是这样，对 schema 和 SQL 的支持仍然是 RDBMS 的独特标志。

MySQL 是最流行的开源关系型数据库。它的设计模块化，提供了可插拔的存储引擎，还支持可插拔的模块来提供额外的特性。从概念层面看，用客户端访问 MySQL 服务器如图 15-1 所示。

MySQL 非常快。对于数千行的数据，它的读写速度通常都非常棒，对大部分用例而言完全够用。数据量增加以后，为服务器提供充足的内存能提升性能。类似大多数数据库，MySQL 在存储引擎的缓存池中保存已获取的行集，再次获取同样的行集时性能明显提升。不过随着数据量增加，SQL 的开销也会变得比较明显。每次获取数据，特别是在频繁并发地由多个客户端发起请求时，都会导致一些高成本的操作：

- 解析 SQL 语句
- 打开表
- 锁表
- 创建 SQL 执行计划
- 解锁表
- 关闭表
- 使用信号量和线程管理并发访问

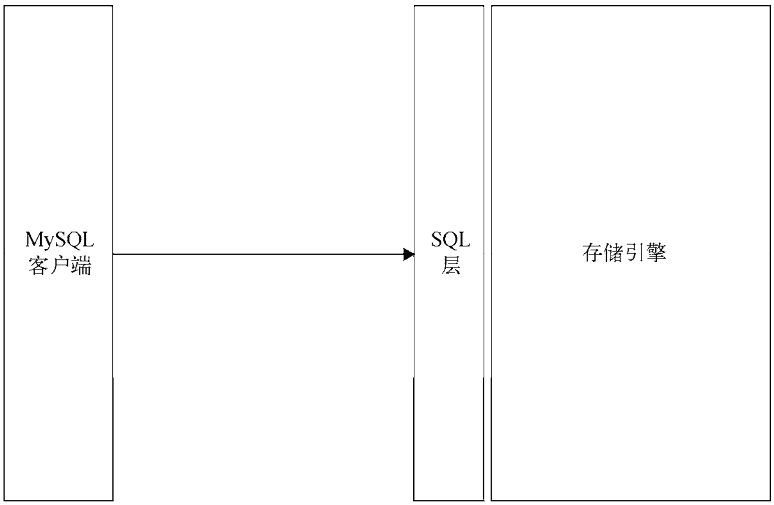


图　15-1

因此，要在大负载下提升性能，必须尽可能缓存更多数据。Memcached 是适用于 MySQL 的典型内存缓存方案。行集缓存在内存里，通过内存提供给客户端。如果有大量内存（比如 32GB），MySQL 和 Memcached 每秒能处理超过 400 000 次查询。当然这些查询都是主键查询，不包括连接等。假设还认为所有相关数据都在内存里，不需要再从磁盘中读取。和内存访问相比，磁盘 I/O 实在是过于昂贵，开销巨大。

前面提到 Memcached 是键/值存储。像 Membase、Redis、Tokyo Cabinet 和 Kyoto Cabinet 也可以和 MySQL 一起使用并达到相近的性能效果。这是一种结合 RDBMS（MySQL）和 NoSQL（如 Memcached）来完成主键查询的场景。图 15-2 描绘了客户端访问 MySQL，Memcached 做前端的典型情景。

MySQL 结合 Memcached 很有效，但架构上存在下面一些缺陷。

- 有两个地方的数据在内存里：存储引擎缓冲区和 Memcached。
- 存储引擎和 Memcached 数据副本的状态可能会不一致。
- 数据通过 SQL 层加载到 Memcached 中，因此 SQL 成本还是存在，只不过最小化了而已。
- 所有数据都在内存里时，Memcached 性能才高，磁盘 I/O 开销可能很大，而且会导致系统变慢。

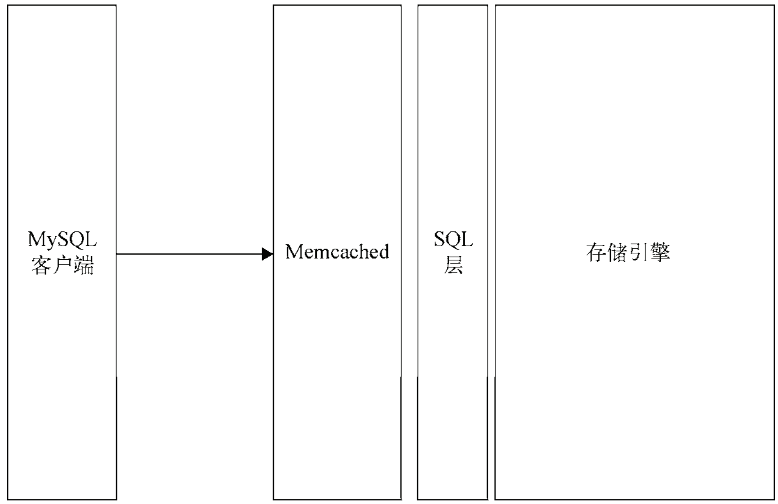


图　15-2

还有一种结合 MySQL 和 Memcached 的方案是越过 SQL 层直接访问存储引擎。MySQL 的 HandlerSocket 插件就是这么做的，它支持绕开 SQL 层直接访问 MySQL 存储引擎，项目开源，github 地址：https://github.com/ahiguti/HandlerSocket-Plugin-for-MySQL。

HandlerSocket 可以被加载到已有 MySQL 服务器上，加载 HandlerSocket 不会关闭 SQL 层。事实上两层都可用。HandlerSocket 为 MySQL 提供了类 NoSQL 的接口，支持更快地访问数据，特别是基于主键的获取。图 15-3 展示的是配置了 HandlerSocket 的 MySQL。

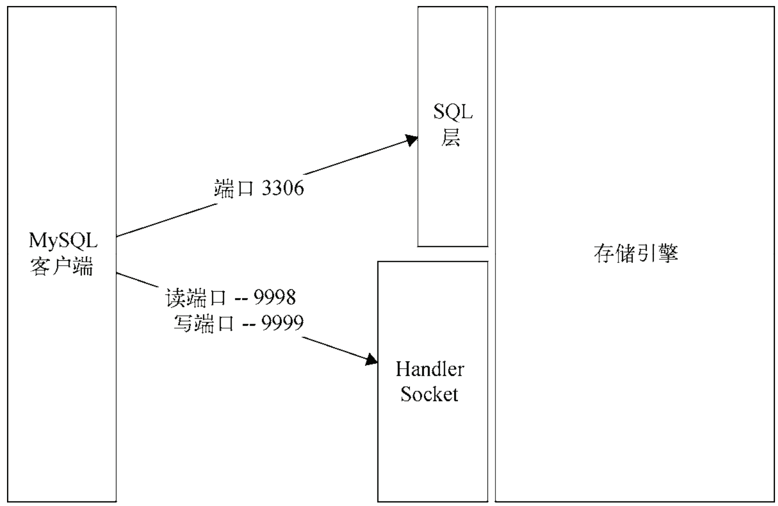


图　15-3

HandlerSocket 实现了网络协议、API 和轻量级的连接来直接访问 MySQL 存储引擎（比如 InnoDB）。它支持和 NoSQL 数据库一样灵活和高性能的查询方式。性能测试数据在这里：http://yoshinorimatsunobu.blogspot.com/2010/10/using-mysql-as-nosql-storyfor.html 。数据显示 HandlerSocket 每秒能完成超过 750 000 次主键查询，其性能之高令人印象深刻。

HandlerSocket 的 API 不会涉及打开、锁住、解锁和关闭表的成本。它的 API 非常轻量级，和 SQL 层相比更偏向 NoSQL。HandlerSocket 自带了 C++和 Perl 的 API。除了核心发布版提供的实现外，还有其他语言的 HandlerSocket API 实现，包括 PHP、Java、Python、Ruby、Javascript 和 Scala。要了解这些类库请访问：https://github.com/ahiguti/HandlerSocket-Plugin-for-MySQL。HandlerSocket 的网络包很小，支持多并发连接。

HandlerSocket 优于 Memcached+SQL 层的地方在于它免去了 SQL 层处理，同时还避免了重复缓存以及复制中潜在的不一致。HandlerSocket 接口直连存储引擎，所以同一个存储不会出现两份副本。

基于 HandlerSocket 的 NoSQL 方案特别适于高性能读的场景。MySQL 引擎提供事务支持和从崩溃中恢复，而且还能用 MySQL 附带的所有工具来监管查询。最后，HandlerSocket 很容易整合进已有的 MySQL 服务器，极为灵活。

除了以 NoSQL 方式使用 MySQL，MySQL 还可以用作最终一致性 NoSQL 存储 Voldemor 的底层引擎。InnoDB 也可以作为存储引擎插入 Riak。

有些情况下，MySQL 也没法作为 NoSQL 用，因为需要的可能是文档数据库、列族存储或键/值存储提供的存储方式。这种时候，可以把 RDBMS 当作事务系统运行，剩下的部分就交给 NoSQL 了。

15.2　静态数据存储

NoSQL 通常都缺乏 RDBMS 提供的事务支持和一致性，有趣的是，这一点常常成为不选择 NoSQL 的一个主要理由。不过在评估数据库的事务支持前，应当首先考虑数据是否变化，是否需要事务支持。

和许多开发者的观念不同，现在很多应用程序需要的事务支持其实非常少，甚至不需要。这主要是因为数据经常是写一次，然后读和处理多次。如果不知道哪些数据算这类，那就打开你的 e-mail 或者社交应用，看看这些系统究竟有多少功能和更新删除有关。许多社交应用里能发送消息或微博，或者更新状态，一般都是写一次，读多次。有部分管理这种活动流的系统能支持更新，不过即便是这些系统，更新往往也不是原地更新（inline update），而是补偿性事务（compensating transaction）。也可能支持删除，但不一定是以交易形式发送给所有接收者。就是说，被删除的微博或者消息可能从源服务器删除了，但不一定从所有消费它的应用里删除了。而且删除也往往是服务发送的一个补偿性消息。

写一次读多次在 RSS 更新、E-mail、SMS 短消息或反馈中普遍存在，征求投票、反馈、评分和评论的应用也通常是写一次读多次的。如果这些应用允许更新，那往往是因为更新并不频繁。

> 前一章我们看到像 HBase、Hypertable 和 MongoDB 这样的 NoSQL 提供行级原子更新，这些数据库并不支持范围的 ACID 事务。不过很多情况下更新之间相互都隔离，不会对一组条目执行，这样行级更新就足够了。
>
> 一部分 NoSQL 数据库例如 Apache Cassandra、Riak 和 LinkedIn Voldemort 都是最终一致性的。也就是说这些数据库不提供基于 Paxos 或者类似算法的强一致性，在更新传递到副本节点过程中会出现不一致，通常在很短时间内就会达到一致。许多应用其实能接受短时间的不一致。

许多巨型社交媒体例如 Facebook、Twitter 和 LinkedIn 都是 NoSQL 和 RDBMS 的大用户。

15.2.1 存储多元化在Facebook中的应用

Facebook 在许多关键应用中都使用 MySQL，同时 Facebook 又大量使用了 HBase。Facebook 在一次技术交流中提到了他们对 MySQL 进行的优化，可以在线访问录音，地址是：www.livestream.com/facebookevents/video?clipId=flv_cc08bf93-7013-41e3-81c9-bfc906ef8442。Facebook 很在乎大容量、高性能，无疑他们对 MySQL 的优化也都是针对这些内容。他们关注于提升每秒的查询次数及控制请求—响应时间的波动。在 2010 年 11 月的一次演讲中，Facebook 曾展示了一些数字，令人印象深刻，他们分享的在线交易处理系统的一些关键指标如下。

- 读响应平均是 4ms，写平均是 5ms。
- 每秒读取的记录最高可达 4.5 亿行，与大多数系统相比，这个数字显然非常大。
- 峰值达到 1300 万次查询/秒。
- 边界情况下能处理 320 万行记录的更新，520 万次 InnoDB 磁盘操作。

虽然 Facebook 每秒查询次数非常惊人，不过他们更关注可靠性。通过主动监测和分析，Facebook 的数据库团队能及时发现服务器性能的问题。慢查询和其他慢速问题会被逐步发现并修正，使得整个系统性能保持良好。更多细节可以从他们的 PPT 中获得。

Facebook 同时也是 Cassandra 的诞生地。最近他们放弃了 Cassandra 而转向 HBase。Facebook 现在的消息系统就是构建在 HBase 上的，它支持每月存储超过 1350 亿消息。Facebook 工程师团队的一份笔记 www.facebook.com/note.php?note_id=454991608919 解释了选择 HBase 的原因。首先是基于 Paxos 的强一致性模型。HBase 的可扩展性很好，也有基础设施能支持高冗余的配置。故障转移和负载均衡默认也都支持，HBase 底层的分布式文件系统 HDFS 又额外提供了一层冗余和容错。此外，还可以复用协调统筹系统 ZooKeeper，并稍作修改来支持用户服务。

因此，对于像 Facebook 这样的公司来说，采取混合策略让他们可以用恰当的工具来处理不同的任务。Facebook 的团队仍然需要修改系统以适应他们的需要，但是他们的故事告诉我们，无论选 DBMS 还是 NoSQL，最重要的都是要选择适合的数据库。Facebook 故事中的另一点是他们选择了自己最熟悉的工具，他们选择的是自己的工程师懂得如何调适的工具，而不是追逐趋势。

例如，坚持使用 PHP 和 MySQL 就为 Facebook 带来了好处，因为工程师们能调整这些工具来满足他们的需要。有人争论老技术不行，不过性能数据清楚地指出 Facebook 已经找到了大规模部署它们的办法。

和 Facebook 一样，Twitter 和 LinedIn 也选择了混合策略。Twitter 积极使用 MySQL 和 Cassandra。Twitter 还使用图数据库 FlockDB 来存储关系，比如谁关注了谁，或者你接受谁的手机提示。过去几年里 Twiiter 变得极为流行，数据量的增长也非常厉害。Kevin Weil 在 2010 年 9 月的演讲上（www.slideshare.net/kevinweil/analyzing-big-data-at-twitter-web-20-expo-nyc-sep-2010）介绍说当时每天的微博和私信量已经达到 12TB 了，假设线性增长，那每年就会增加超过 4PB。随着越来越多的人使用 Twitter 沟通，Twitter 的数据必定会继续增长，越变越大。要处理这么多数据一定是巨大的挑战。Twitter 用 Hadoop、MapReduce 和 Pig（http://pig.apache.org/）进行大数据分析。Pig 语句最终会落地为 Hadoop 集群上的 MapReduce 任务。在 Twitter，大量核心存储仍然依赖 MySQL，Twitter 的很多功能都大量使用 MySQL。Cassandra 只在特定场景中使用，比如存储地理信息数据。

与 Twitter 相似，LinkedIn 也依赖多种不同的数据存储。Jay Kreps 在去年的 Hadoop 峰会上介绍了 LinkedIn 的大数据架构和存储，幻灯片地址为：www.slideshare.net/ydn/6-data-application-linkedinhadoopsummmit2010。LinkedIn 用 Hadoop 来处理大规模分析任务，比如推测你可能认识的人。Hadoop 集群上的数据量很大，范围约为每天 1200 亿个关系，涉及 82 个 Hadoop 任务和超过 16TB 中间结果数据。概率图被从离线存储复制到一个在线的 NoSQL 集群里，使用的 NoSQL 数据库是 Voldemort（类似 Apache Dynamo），它将数据表示为键/值对。关系图是只读的，所以 Voldemort 的最终一致性模型不会引起任何问题。数据是批量处理的，但实时搜索过滤，过滤器可能会排除某人不认识的那些人。

通过 Facebook、Twitter 和 LinkedIn，我们可以很明显地看到，混合策略有它的好处，可以优化软件栈，方便特定场景用适合的数据库。

15.2.2　数据仓库和商业智能

有这样一类软件，专门用来存储和处理已归档的数据。通常这些数据仓库构建在已有的事务数据之上，这些数据又叫事实数据。通过对数据仓库里的数据进行分析处理来揭示模式，或者揭秘趋势。所有归档的数据都是只读的，几乎不需要事务。这些数据一般放在专门的数据存储中，它们能保存大量数据，按照多维度进行分析。

有了 Hadoop 以后，一部分大规模分析就通过 MapReduce 任务来完成。有了众多项目的帮助，MapReduce 分析模型正在逐渐丰富起来，这些项目包括查询工具 Hive，工作流定义高级语言 Pig 等。此外，有关 MapReduce 的创新仍在继续。Apache Mahout 项目在 Hadoop 上建立机器学习的基础设施，使人们可以通过 Mahout 在 Hadoop 的 MapReduce 基础设施上运行协同过滤或者朴素贝叶斯分类器。

15.3 Web 框架和 NoSQL

创建可扩展的 Web 应用非常有挑战性。需求不断变化，数据在持续演进。这种情况下传统的 RDBMS 就显得有些不太灵活了。有些时候，可能文档数据库更适合。

15.3.1 Rails和NoSQL

Ruby on Rails 不用多说，它是目前为止最流行的敏捷 Web 开发框架。由于坚持约定优于配置，Rails 让 Web 开发变得轻松有趣。它实现了 MVC 框架，RESTful 动词是底层模型基础的默认操作，ActiveRecord 使得模型对象能自动映射到关系表中的持久化数据中。视图提供界面来操纵底层数据，控制器协调模型和视图。

如果你用 Rails 来开发，很容易就能用 MongoDB 替换 MySQL、PostgreSQL 或任何其他 RDBMS。mongo_mapper 提供了对 MongoDB 的巨大支持。

要在 Rails 中使用 MongoDB，首先关闭 ActiveRecord。使用 MongoDB 时，不需要 ORM 层。在多数平台上安装 Rails 首先要有 Ruby 和 RubyGems，然后执行下面命令：

```
gem install rails
```

安装成功以后，很容易创建新应用并指定它不使用 ActiveRecord。创建不带 ActiveRecord 的 Rails 应用命令如下：

```
rails new sample_app -skip-active-record
```

下面，安装 mongo_mapper，它以 gem 形式发布，安装命令跟上面差不多：

```
gem install mongo_mapper
```

安装完 mongomapper 后，将其添加到 gemfile 里，这样 bundler 就可以为 Rails 应用提供 mongomapper 了。修改 gemfile 如下：

```
require 'rubygems'
require 'mongo'
source 'http://gemcutter.org'

gem "rails", "3.0.0"
gem "mongo_mapper"
```

为了避免 bson_ext 问题，在 gem 定义前要先准备好 Mongo 和 rubygems。

下面，运行 bundler install 来下载安装需要的 gem。

bundler 准备好以后，在 `config/initializers` 下面创建一个初始化文件，并向其中添加下列内容：

```
MongoMapper.connection = Mongo::Connection.new('localhost', 27017)
MongoMapper.database = "#sample_app-#{Rails.env}"

if defined?(PhusionPassenger)
   PhusionPassenger.on_event(:starting_worker_process) do |forked|
     MongoMapper.connection.connect if forked
   end
end
```

下面是利用 mongo_mapper 的一个简单例子：

```
class UserData
   include MongoMapper::Document

   key :user_id, Integer
   key :user_name, String
end
```

下面用控制器来持久化模型对象：

```
class MyActionController < ApplicationController
  def create_user

    @auser = UserData.create(
            {
              :user_id => 1,
              :user_name => "John Doe",
              :updated_at => Time.now
            })
    @auser.save()
  end
end
```

这个操作可以通过 REST 风格的 URL 触发：

```
get 'my_action/create_user'
```

除 Rails 外的其他 Web 框架还包括 Django（Python）和 Spring（Java）。

15.3.2　Django和NoSQL

Django 之于 Python 社区，就像 Rails 之于 Ruby 开发者。Django 是轻量级 Web 框架，支持原型和快速开发。Django 也是习惯通过 ORM 将模型映射到数据库。SQL 标准和 ORM 层的存在使得 Django 可以替换不同的 RDBMS。但是替换 NoSQL 不常有。事实上，针对特定一种 NoSQL 编写的代码是如此特殊，常常不能用在另一种 NoSQL 产品上。其实大家都希望应用程序能无缝支持各种 NoSQL 产品，又同时支持 SQL 和 NoSQL 产品。

不同 NoSQL 产品的索引以及连接数据的方式各有不同。要让 Django 应用支持多种 NoSQL 产品，就要通过为每种 NoSQL 各自编写索引管理和数据聚合代码。

最后一个要点是，我们知道云平台上 NoSQL 很流行，但是要在这些平台之间移植相当有难度。例如，GAE（Google App Engine）在 Google Bigtable 基础上提供建模抽象，而 AWS（Amazon Web Service）则通过 SimpleDB 提供托管的文档数据库。为 GAE 或者 AWS 编写的 Django 应用会和平台紧紧捆绑在一起，这样做会让两个平台之间迁移或者迁移到其他平台都变得极其困难。有时候，这种迁移几乎要求重写应用，导致厂商锁定，并增加平台迁移的财力和人力成本。

django-nonrel 独立开源项目是为了解决这些问题而建立的，它为 Django 提供公共抽象层来支持多种 NoSQL 产品。项目源码地址：https://bitbucket.org/wkornewald/django-nonrel/src。Waldemar Kornewald 和 Thomas Wanschik 是该项目的创建人和核心贡献者。

django-nonrel 对 Django 核心的修改刚够支持 RDBMS 以外的数据库，更繁重的工作则交给了 django-dbindexer，它负责去规范化和连接数据集。

django-dbindexer 是一个尚处于早期阶段的开源项目，地址：https://bitbucket.org/wkornewald/django-dbindexer/src。它的定位是 NoSQL 数据库的封装层，主要处理各种 NoSQL 产品的差别，区分大小写的查询和连接（join）都是在这层处理的。例如，MongoDB 里不区分大小写的过滤不能用索引。而全表扫描比索引效率低。在 django-dbindexer 这一层，这种低效过滤就可以被当作是区分大小写的过滤，从而利用到索引。

缺失 RDBMS 里 SQL 这样强大的公共查询语言，使得在 NoSQL 平台上支持某些查询变得很有难度。django-dbindexer 简化和标准化了查询 API。所以下面的 GAE 代码：

```
# models.py:

class MyModel(models.Model):
    name = models.CharField(max_length=64)
    lowercase_name = models.CharField(max_length=64, editable=False)
    last_modified = models.DateTimeField(auto_now=True)
    month_last_modified = models.IntegerField(editable=False)

    def save(self, *args, **kwargs):
        self.lowercase_name = self.name.lower()
        self.month_last_modified = self.last_modified.month
        super(MyModel, self).save(*args, **kwargs)
def run_query(name, month):
    MyModel.objects.filter(lowercase_name=name.lower(),
                           month_last_modified=month)
```

models.py

可以被替换成更优雅、更干净和可重用的代码：

```
# models.py:

class MyModel(models.Model):
    name = models.CharField(max_length=64)
    last_modified = models.DateTimeField(auto_now=True)

def run_query(name, month):
    MyModel.objects.filter(name__iexact=name, last_modified__month=month)

# dbindexes.py:

from models import MyModel
from dbindexer.api import register_index
register_index(MyModel, {'name': 'iexact', 'last_modified': 'month'})
```

models_with_dbindexer.py

更多内容可以阅读 django-dbindexer 的文档，地址：www.allbuttonspressed.com/projects/django-nonrel。

15.3.3　使用Spring Data

虽然在敏捷开发中 Rails 和 Django 算是比较流行的 Web 框架，但是仍然有很多企业开发者使用 Java 来构建新的应用。Spring 是广泛采用的 Java 依赖注入框架，它通过 Spring Data 项目加入了对 NoSQL 的支持。有关 Spring Data 项目的信息可以访问：www.springsource.org/spring-data。Spring Data 不仅为许多 NoSQL 提供抽象层，同时还简化了基于 MapReduce 的处理以及访问云平台。

下面我介绍一个小例子用 Spring Data 访问 Redis。Spring Data 通过子项目支持 Redis，并用类似方法支持其他 NoSQL。用来和 Redis 交互的 Java 客户端类库早就有了，比如 JRedis（http://code.google.com/p/jredis/）、jedis（https://github.com/xetorthio/jedis）。Spring Data 通过 RedisTemplate 对这些 Java 客户端类库进行抽象，这类似于 Spring 用 JdbcTemplate 抽象掉 JDBC，用 HibernateTemplate 抽象掉 Hibernate，其目的是对开发者隐藏底层 API 细节。

首先下载并安装 Spring Data Redis 子项目，地址：https://github.com/SpringSource/spring-data-keyvalue。为了简单和更快速地开发，你可以使用 STS（SpringSource Tool Suite，SpringSource 工具套件），用项目模板来创建一个新的 Spring 项目，STS 地址是：www.springsource.com/developer/sts。STS 用 Maven 配置和构建项目，因此定义都声明在项目的 POM（project object model，项目对象模型）里。更多有关 Maven 的内容可以访问 http://maven.apache.org/。修改 `pom.xml` 文件来配置它就是要使用 Spring Data Redis 子项目。代码清单 15-1 列出了典型的 `pom.xml` 文件内容。

代码清单 15-1　Spring Data Redis 项目 POM

```
<?xml version="1.0" encoding="UTF-8"?>
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.treasuryofideas.pronosql</groupId>
  <artifactId>redis</artifactId>
  <name>redis-dictionary</name>
  <packaging>war</packaging>
  <version>1.0.0-BUILD-SNAPSHOT</version>
  <properties>
    <java-version>1.6</java-version>
    <org.springframework-version>3.0.5.RELEASE</org.springframework-version>
    <org.springframework.roo-version>1.0.2.RELEASE</org.springframework.roo-
version>
    <org.aspectj-version>1.6.9</org.aspectj-version>
    <redis.version>1.0.0.M2-SNAPSHOT</redis.version>
  </properties>
  <dependencies>
    <!-- Spring -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>${org.springframework-version}</version>
```

```
      <exclusions>
        <!-- Exclude Commons Logging in favor of
<span class="hiddenSpellError"
pre="of ">SLF4j</span> -->
        <exclusion>
          <groupId>commons-logging</groupId>
          <artifactId>commons-logging</artifactId>
        </exclusion>
      </exclusions>
    </dependency>

    <!-- <span class="hiddenSpellError"
pre="">AspectJ</span> -->
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjrt</artifactId>
      <version>${org.aspectj-version}</version>
    </dependency>

    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.15</version>
      <exclusions>
    <exclusion>
      <groupId>javax.mail</groupId>
      <artifactId>mail</artifactId>
    </exclusion>
    <exclusion>
      <groupId>javax.jms</groupId>
      <artifactId>jms</artifactId>
    </exclusion>
    <exclusion>
      <groupId>com.sun.jdmk</groupId>
      <artifactId>jmxtools</artifactId>
    </exclusion>
    <exclusion>
      <groupId>com.sun.jmx</groupId>
      <artifactId>jmxri</artifactId>
    </exclusion>
  </exclusions>
  <scope>runtime</scope>
</dependency>

<!-- @Inject -->
<dependency>
  <groupId>javax.inject</groupId>
  <artifactId>javax.inject</artifactId>
  <version>1</version>
</dependency>

<!-- Test -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
```

```
    <version>4.8.1</version>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <version>${redis.version}</version>
  </dependency>

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-keyvalue-core</artifactId>
    <version>${redis.version}</version>
  </dependency>

  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>${org.springframework-version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
      <version>${org.springframework-version}</version>
    </dependency>

    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-io</artifactId>
      <version>1.3.2</version>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>${org.springframework-version}</version>
      <scope>test</scope>
      <exclusions>
        <exclusion>
          <groupId>commons-logging</groupId>
          <artifactId>commons-logging</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
  </dependencies>
  <repositories>
    <repository>
      <id>spring-maven-milestone</id>
      Springframework Maven Repository
      <url>http://maven.springframework.org/milestone</url>
```

```
    </repository>
    <repository>
      <id>spring-maven-snapshot</id>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
      Springframework Maven SNAPSHOT Repository
      <url>http://maven.springframework.org/snapshot</url>
    </repository>
  </repositories>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>${java-version}</source>
          <target>${java-version}</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>install</id>
            <phase>install</phase>
            <goals>
              <goal>sources</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </project>
```

TagSynonyms

Maven 让构建项目、定义依赖和管理依赖的工作变得简单轻松。前面的 POM 文件定义了构建项目所需的所有外部依赖。用 POM 文件管理项目生命周期时外部库就会下载并配置好。

下面用 Redis 和 Spring Data 写一个简单例子。这个例子是构建一个标签同义词表。在这个表里，一个标签作为键，所有含义相同或近似的标签组成值。比如“Web”作键，“internet”和“www”作值，这种表用 Redis 列表很容易实现。要通过 Spring 访问这个存储，需要先创建 DAO 类，如代码清单 15-2 所示。

15

代码清单 15-2 TagSynonymsDao

```
import org.springframework.data.keyvalue.redis.core.RedisTemplate;

public class TagSynonymsDao {

  private RedisTemplate<String, String> template;
```

```
  public TagSynonymsDao(RedisTemplate template) {
    this.template = template;
  }

  public Long addSynonymTag(String keyTag, String synonymTag) {
    Long index = template.opsForList().rightPush(keyTag, synonymTag);
    return index;
  }

  public List getAllSynonymTags(String keyTag) {
    List<String> synonymTags = template.opsForList().range(word, 0, -1);
    return synonymTags;
    }

    public void removeSynonymTags(String... synonymTags) {
    template.delete(Arrays.asList(synonymTags));
}

}
```

TagSynonyms

上述程序通过标签同义词表演示了 RedisTemplate。数据访问类通过模板与 Redis 交互，用 RedisTemplate 里定义的方法来插入元素、执行范围查询或删除元素。

例子到此为止，希望通过例子能让你对这个抽象层有一些感觉，它可以支持不同的 NoSQL，并让你在 RDBMS 和 NoSQL 之间平滑切换。

15.4　从 RDBMS 迁移到 NoSQL

从结构化的 schema 迁移到弱 schema 形式不算太难，很多时候就是把 RDBMS 表里的数据导入 NoSQL 集合而已。不过如果 NoSQL 数据库是列族有序存储，或者键/值存储，那就复杂多了。更改范式往往意味着重新设计。

更大的问题是专门查询和次级索引，这些在 NoSQL 环境里往往不容易实现。NoSQL 更是从查询的角度来看数据，而不是从通用存储来看。

为了便于从 RDBMS 导入数据到 Hadoop 中进行 NoSQL 风格的处理，Cloudera 创建了一个开源项目 Sqoop。Sqoop 是命令行工具，有下列功能。

- 将单个 RDBMS 表或整个数据库导入 HDFS 的文件中。
- 生成 Java 类来访问被导入数据。
- 支持从 SQL 数据库直接导入 Hive 数据仓库。

更多 Sqoop 内容请访问：https://github.com/cloudera/sqoop。

15.5　小结

本章介绍不同种类数据库共存的主题，展示同时使用 RDBMS 和 NoSQL 的方法。文中引用

了大型社交媒体的例子以便从中汲取灵感，Facebook、Twitter 和 LinkedIn 的例子都有所提到。

随后简要介绍弥补了 RDBMS 和 NoSQL 之间空白的产品，并通过例子介绍了流行框架 Rails、Django 和 Spring 对 RDBMS 和 NoSQL 的支持。

最后，简要谈到从 RDBMS 迁移到 NoSQL 的话题，以展示数据可以从表中导入到更适于 MapReduce 风格分析的结构中。

下一章介绍性能调校的话题。

第 16 章

性能调校

本章内容

- ❑ 理解影响并行可扩展应用的因素
- ❑ 优化并行处理，特别是利用 MapReduce 模型的处理
- ❑ 展示一组并行处理的最佳实践
- ❑ 演示一些 Hadoop 性能调校小秘诀

今天，NoSQL 中主要的大数据分析都是在 MapReduce 模型上进行的处理。Hadoop 就构建在这个模型上，而且每种支持海量数据的 NoSQL 产品都利用了这个模型。本章我们了解对 MapReduce 风格大数据处理和相关应用的调优。当然本章提供的不是标准解决方案，而是一些在调优并行可扩展应用时值得记住的重要概念和实践。任何优化问题都是和它的需求、场景息息相关的，所以提出一个适用于所有情况的通用方案可能不太可行。

16.1　并行算法的目标

MapReduce 极大地简化了可扩展并行处理。通过坚持在并行线程或进程间不共享数据，MapReduce 因此而创造出一种无瓶颈的方式，能随着负载增加不断向外扩展。本质上目标永远都是减少延迟和增加吞吐。

16.1.1　减少延迟的含义

简单说减少延迟就是减少程序执行时间。程序完成得越快（给定输入得到计算结果的耗时越少），延迟就越小。输入输出不变的情况下，减少延迟通常需要选择最优算法，并行执行子任务。如果一个任务能分解成一组独立的、可并行执行的任务，那么并行执行可以减少完成总任务的耗时。因此在并行程序中，延迟实际上可通过最小“原子”任务的耗时来度量。“原子”的含义是说工作单元无法进一步分解成并行子任务。如果不能并行化，那延迟就是执行整个程序的耗时。

在优化算法的同时，需牢记算法要符合 map 和 reduce 函数模型。当然，如果需要，这些函数可以遍历多次。

16.1.2 如何增加吞吐

吞吐是指给定进程（过程）可以处理（并产生结果）的输入量。对大数据，吞吐往往更重要，有时甚至牺牲延迟，因为分析大数据可不是小事。比如，Twitter 的 Kevin Weil 在 2010 年 Web 2.0 Expo 的演讲上（www.slideshare.net/kevinweil/analyzing-big-data-at-twitter-web-20-expo-nyc-sep-2010）提到 Twitter 每天的微博量达到 12TB。如果磁盘写入速度只有 80Mbps，那么多数据要 48 小时才能写完。这样的故事到处都在发生，比如 Facebook 和 Google，他们的用户流量每天也会产生大量数据。

Hadoop 支持分析大数据，甚至大到一台机器都放不下。在传统的单机系统中，吞吐常受可用资源限制。比如 RAM 的大小或是 CPU 的数量或频率决定了一台机器的处理量。只要数据持续增长下去，再强力的机器也会达到上限。在利用 Hadoop 分布式文件系统（HDFS）的环境中，这种限制不再是那么严重的问题。集群增加节点就能处理更多数据。并行化的一个副作用是普通的商业硬件（和更强的机器相比）能够帮助有效地增加吞吐。

16.1.3 线性扩展

在基于 MapReduce 的模型中，处理通常是并行的，而扩展是线性的。如果集群一个节点每秒能处理 x MB 数据，则 n 个节点每秒能处理 n 乘 x MB 数据。反过来说，就是数据每增加 x MB，保持同样的处理速度就要增加一个新节点。另外，如果所有节点负载都一样，那么只要增加负载同时增加新节点，处理时间就能保持不变。如果数据不变，增加节点，则处理时间等比例减少。

数学表达如下。

- 单节点处理数据量 y 耗时=t 秒。
- N 节点处理数据量 y 耗时=t/n 秒。

公式假定任务可以分成同等的工作单元，每个单元处理时间大致相同。

16.2 公式与模型

Hadoop 的关键贡献者之一 Milind Bhandarkar 在有关扩展 Hadoop 应用的主题演讲中用三个著名公式进行了总结。

- Amdahl 法则
- Little 法则
- 消息成本模型

16.2.1 Amdahl 法则

Amdahl 法则用来找出系统局部改进对整体性能的最大提升。Amdahl 法则取名自 Gene Amdahl（www.computer.org/portal/web/awards/amdahl），著名计算机架构师，曾为创造 IBM 大型机作出了贡献。

下面用例子扼要解释 Amdahl 法则。假设某个程序运行需要 5 小时，除一小部分以外，剩下部分都能并行执行，不能并行执行的这部分耗时 25 分钟，那么它就定义了整个程序能达到的最佳速度。基本上，程序的线性部分限制着程序的性能。

用数学方式表达这个例子如下。

- 程序总执行时间：5 小时（300 分钟）。
- 顺序执行时间：25 分钟。
- 程序可并行部分：约 91.6。
- 程序不可并行部分（或本质上是顺序的）：8.4。
- 因此，并行版比非并行版最多提升速度 $1 / (1 – 0.916) \approx 11.9$。

换句话说，同一个程序，完全并行化的版本比没有并行化的版本快 11 倍。

Amdahl 法则将提升速度的计算概括为如下表达式：

$$1 / ((1 – P) + P/S)$$

其中 P 表示并行化的部分，S 是并行化部分与非并行化部分性能之比。

该等式还考虑了程序不同部分提速不同的情况。比如，一个程序可以并行化为四部分：$P1$、$P2$、$P3$ 和 $P4$，分别占比 10%、30%、40%和 20%。如果 $P1$ 增速 2 倍、$P2$ 为 3 倍、$P3$ 为 4 倍，$P4$ 无增速，则整体运行时间如下：

$$0.10/2 + 0.30/3 + 0.40/4 + 0.20/1 = 0.45$$

因此最大增速为 $1 / 0.45 = 2.22$，超过原来程序 2 倍。

更多有关 Amdahl 法则的内容请参阅：www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf。

Amdahl 法则对 MapReduce 和多核编程同等适用。

> Gustafson 法则（http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.85.6348）重新评估了 Amdahl 法则并指出：同样的时间里，提供更多计算资源就能解决更复杂的问题，越简单的问题消耗资源越少。因此，Gustafson 法则与程序线性部分对可扩展性限制的理论相矛盾，特别是在用更多计算资源执行复杂、有重复性的大型任务时。

16.2.2　Little 法则

Little 法则适用于并行计算，源自经济学和队列理论。它表面上看起来很简单，不过却提供了概率分布无关的方式来分析稳定系统的负载。法则指出：稳定系统的平均客户数是平均到达率与客户在系统中停留时间的乘积。公式如下：

- $L *= kW*$
- L 是稳定系统的平均客户数
- k 是平均到达率
- W 是客户在系统中的停留时间

为了进一步理解，假设有个系统，比方说一个加气站，只有一个收款台，只收现金。如果每小时有 4 位顾客进站加气，每位顾客在加气站停留 15 分钟（0.25 小时），则任意时刻加气站平均有 1 位顾客。如果同一时刻到达加气站的顾客数超过 4 个，很明显系统会出现瓶颈。如果顾客因为等待时间过长，不耐烦而离开（没有加气），很可能退出率大于到达率，这种情况下系统将变得不稳定。

从 Little 法则的观点来看系统，把进程看作是顾客，将其转换成并行程序后，需要消耗一定的时间 W 才能完成执行，系统任何时间最多只能容纳 L 个进程，则定长时间内进程的到达率不能超过 L/W。如果到达率超了，系统就会阻塞，计算耗时和系统容量都会受到影响。

16.2.3　消息成本模型

第 3 个等式是消息成本模型。消息成本模型把端到端的消息发送成本分解成固定成本和滑动成本。简单说来，消息成本模型的公式如下。

$$C *= *a *+ *bN$$

- C 是消息从 A 端发送到 B 端的成本。
- a 是发送消息的前置成本。
- b 是消息每字节的成本。
- N 是消息的字节数。

这个等式很简单，其中有两个关键点：

- 无论消息有多大，凡传输必涉及固定成本。对消息来说，建立连接、握手和配置的成本是很常见的。
- 传输本身的成本与消息的大小线性相关。

消息成本模型对网络传输消息提供了一些很有趣的见解。在千兆以太网上，a 大概是 300 微妙，即 0.3 毫秒，b 是 125MB 每秒。1 Gigabit 是 1000Mb（125MB）。千兆以太网传输速率为 125MBps。每秒 125MB 的成本和每毫秒 125KB 的成本相同，因为 1000 毫秒是 1 秒，1000KB 是 1MB。也就是说 100 个 10KB 的消息需要 $100 \times (0.3 + 10 / 125)$毫秒，即 38 毫秒，而 10 个 100KB 的消息只需要 $10 \times (0.3 + 100 / 125)$毫秒，即 11 毫秒。因此，一种优化成本的方法就是每次发送的消息要尽可能大，这样固定成本就被分摊到更大的消息上。

> 理论计算时，消息成本模型中的固定成本 a 对不同大小的消息都一样，但实际上 a 值会受消息大小影响。

16.3　分区

并行计算中很重要的一点就是分区。在 MapReduce 中，每个 reducer 形成一个分区。map 阶段生成的键/值对被 reducer 消费。由于 MapReduce 选择无共享的处理模型，因此键相同的键/值

对必须去同一个分区，并由同一个 reducer 处理。

Hadoop MapReduce 框架中定义了默认的 partitioner：HashPartitioner. HashPartitioner 利用键的 hashCode 方法的返回值来分区。也就是说，“hashCode 值”模“分区数”= n（n 用来将键/值对分发到不同分区上）。

Hadoop 用接口 Partitioner 来决定 map 任务生成的键/值对分发到哪个分区。分区数即 reduce 任务数，在任务启动时是已知的。Partitioner 接口如下：

```
public interface Partitioner<K, V> extends JobConfigurable {
  int getPartition(K key, V value, int numPartitions);
```

getPartition 方法接受键、值和分区数作为参数，返回一个分区号，此分区号标识了键/值对应被分发到的分区。对任意两个键 k1 和 k2，如果 k1 和 k2 相等，getPartition 返回的分区号也相等。

如果根据键/值对进行分区不均衡，就可能出现负载不均衡或过度分区的情况，两者都低效。如果少数 reducer 处理大部分负载，而剩下的空闲，就是负载不均衡。不均衡会增加延迟。机器和磁盘在满负荷下容易变得更慢，或触发边界条件降低效率。负载不均衡就会导致部分 reducer 变成满负荷状态。

从前面的 Amdahl 法则我们知道，任何并行优化都受制于最长的顺序任务。在 MapReduce 处理中，较长时间执行会导致瓶颈。它也会导致后续的等待，因为 reduce 和 grouping 任务只有处理完所有键/值对以后才完成整个处理。

16.4　规划异构环境

Hadoop 默认的规划算法通过比较每个任务的进度和平均进度来规划任务。默认的 scheduler 做了下列假设：

- 节点工作速率不变；
- 任务执行的吞吐不变。

在异构环境中，默认 scheduler 算法不够优化，因此需要专门为异构环境做一些改进。

LATE（Longest Approximate Time to End）scheduler 在默认 Hadoop scheduler 上做了改进。LATE scheduler 只在较快的节点上执行 speculative 任务，它还限制了被 speculate 的任务数。此外还有一个慢任务阀值来决定一个任务是否慢到了要 speculate。

虽然 LATE scheduler 对默认的 scheduler 做了改进，但这两个 scheduler 都是静态地计算任务进度。SAMR（自适应 MapReduce 规划算法）的性能超过了它们。更多有关 SAMR 的内容可以阅读论文“SAMR: A Self-adaptive MapReduce Scheduling Algorithm in Heterogeneous Environment”，作者是 Quan Chen、Daqiang Zhang、Minyi Guo、Qianni Deng 和 Song Guo. 论文在线地址：http://portal.acm.org/citation.cfm?id=1901325。

16.5 其他 MapReduce 调校

有很多配置参数会影响 MapReduce，可以通过合理地配置它们来实现更好的性能。

16.5.1 通信成本

有时数据量太大，MapReduce 算法复杂度不是那么重要，这时的关注点更集中在如何开始处理大数据上。但是请记住，尽可能地移除 reduce 任务可以最小化一些通讯成本和相关的算法复杂度。这种情况下 map 阶段完成所有事情。如果不可能消灭所有 reduce 任务，那么在所有 map 任务完成前启动 reduce 任务也可以提升性能。

16.5.2 压缩

当数据在节点间，在 map 和 reduce 任务间传输时，对数据进行压缩能显著改善性能。这样能减少通讯成本，减少带宽和网络使用。对大型集群和较大的任务，压缩能带来很多好处。

> 有些数据不易压缩，或者压缩不足以提供很多好处。

打开压缩就是将一个配置项设为 true 这么简单。这个配置项是：

```
mapred.compress.map.output
```

压缩算法也可配置。用 `mapred.map.output.compression.codec` 来配置。

> LZO 是适用实时压缩的算法，它偏重压缩速度而非压缩率。更多有关 LZO 算法内容参阅：www.oberhumer.com/opensource/lzo/。

更进一步的改善是使用可分割 LZO 算法。绝大部分 MapReduce 任务都是 I/O 密集的。如果 HDFS 上的文件能压缩成一种可分割的，又能直接被 MapReduce 任务消费的格式，IO 和整体性能就都可以得到改善。用普通 gzip 压缩算法并行分割 gzip 段会产生问题，所有分割部分必须由同一个 mapper 处理。如果只用一个 mapper，那并行化就会受到影响。bzip2 避免了这个问题，分割部分可以发送给不同的 mapper，但是解压缩是 CPU 密集的，因此 I/O 上省下来的时间又耗在 CPU 上。LZO 是比较好的折中，大小和解压速度都不错。更多有关可分割 LZO 算法的内容请参阅：https://github.com/kevinweil/hadoop-lzo。

16.5.3 文件块大小

Hadoop 的底层分布式文件系统 HDFS 支持存储非常大的文件。HDFS 块大小默认是 64MB。如果集群比较小，但是数据又很大，用默认块大小会生成大量 map 任务。例如 120GB 输入会创

建 1920 个 map 任务。计算如下：

$$(120 * 1024) / 64$$

因此，较小的集群可以适当提高块大小。当然别提得太高，以至于集群所有节点都用不上了。

16.5.4　并行复制

map 的输出通过复制传给 reducer。如果 map 任务的输出很大，就可以用多个线程并行复制。增加线程会增加 CPU 使用率，但能减少延迟。默认线程数是 5。可以设置下面这个配置项来增加线程数：

```
mapred.reduce.parallel.copies
```

16.6　HBase Coprocessor

HBase coprocessor 的灵感来自 Google Bigtable 的 coprocessor。一些简单的处理，例如计数、聚合之类可以放在服务器端进行以提高性能。Coprocessor 就实现了这个想法。

HBase 中有三个接口实现了 coprocessor 框架，它们分别是 Coprocessor、RegionObserver 和 Endpoint。通过重载 Coprocessor 和 RegionObserver 的方法来插入用户代码，coprocessor 框架负责调用这些方法。能通过加载多个 Coprocessor 或 RegionObserver 来扩展方法。它们链接起来顺序调用，coprocessor 的调用顺序是由优先级决定的。

通过服务器端的端点和客户端类库提供的动态 RPC，你可以自定义扩展客户端和 region server 之间的 RPC 事务。

16.7　布隆过滤器

布隆过滤器在第 13 章介绍过。如果不清楚它的定义，可以先回顾一下。

目前在 HBase 中，获取行记录的调用会从 region 的所有 StoreFile 中并行获取行记录，导致 *N* 次磁盘读取。布隆过滤器提供轻量级内存结构来减少磁盘读取，从读取 *N* 次到只是读取很可能包含行记录的文件。

由于读取是并行的，所以对单个 get 的调用在性能上没有多少帮助。而且读的性能取决于磁盘读的延迟。如果把并行 get 调用换成顺序调用，布隆过滤器对读取延迟的影响就很明显了。

布隆过滤器有可能比数据更大，这是默认不开启它的一个重要原因。

16.8　小结

本章介绍了一些 MapReduce 并行处理的性能调优技术。MapReduce 算法使得我们可以用普通商业机来处理大量数据。扩展 MapReduce 算法需要一些配置。优化的 MapReduce 配置可以提高性能。

本章展示的性能调优方法是通用的，只是用了 Hadoop 及相关工具来演示。

第 17 章 工具和实用程序

本章内容

- 了解监测管理 NoSQL 的各种工具
- 了解日志处理、MapReduce 管理及搜索相关工具
- 演示与管理可扩展性、健壮性有关的工具

这本书的主题是 NoSQL，最初的目标帮助你熟悉领域，而非成为某种 NoSQL 产品专家。为此我尽可能多地介绍了相关的概念和不同类型的 NoSQL 产品。现在这个目标已经达到，你得到了足够多的材料，对这个发展中领域的基本构成也有了一定的把握和信心。最后一章我们再加一把油。本章不再介绍更多的概念，而是介绍一些既有趣又重要的工具和实用程序，使用 NoSQL 时它们可能派得上用场。我给出的工具列表不是很完备，也不是最佳工具集合，只是比较有代表性。

本章内容围绕着 14 种不同的工具或实用程序来组织，它们都是开源的，并且可以随意获取。尽管这些工具都与 NoSQL 相关，不过它们之间是相互独立的。这意味着你可以从头顺序读到尾，也可以直接翻看介绍某个工具的特定页。

前几种工具，特别是 RRDTool 和 Nagios，不仅对 NoSQL 系统有用，其监控与管理功能对所有类型的分布式系统也都很有用。

17.1 RRDTool

RRDTool 是开源工具，用来做高性能日志和进行时间序列数据的图形化。它能很好地继承命令行脚本及许多脚本语言，包括 Python、Ruby、Perl、Lua 和 Tcl。RRDTool 用 C 编写，可以编译到大部分平台上，支持在 Linux、Windows 和 Mac OS X 上运行。下载地址：http://oss.oetiker.ch/rrdtool/。

RRDTool 包含数据库和图形生成与渲染环境。RRDTool 数据库不同于传统 RDBMS，更像滚动截断的日志文件。RRDTool 非常适于监控，因为它能捕获和图形化各种性能和利用率指标。

下面用一个简单例子展示 RRDTool。假设你需要捕捉一台运行 NoSQL 数据库的机器的 CPU 利用率，每 60 秒捕捉一次。此外，你还想每小时计算一下平均利用率，计算结果保存一天（24 小时）。用 RRDTool 保存和分析这类数据非常容易。

RRDTool 数据库这种存储方案好比绕圆环。数据沿圆周写入，最终一定会回到起点。回到起点后，新数据会覆盖老数据。因此可以存储的数据总量早就由数据库存储总量决定好了，类似圆的周长是预先定好的。

要创建 RRDTool 数据库，最简单的方式是用命令行接口（CLI），基本的 CPU 利用率的例子可以表示如下：

```
rrdtool create myrrddb.rrd \
        --start 1303520400 \
        --step 60 \
        DS:cpu:GAUGE:120:0:100 \
        RRA:AVERAGE:0.5:60:24 \
        RRA:AVERAGE:0.5:1440:31
```

这条命令创建了名为 `myrrddb.rrd` 的 RRDTool 数据库。创建数据库时会初始化一些属性，这些属性定义要捕捉的指标以及如何汇总这些指标。下面逐行解析，以便你理解所有参数。

参数 `start` 和 `step` 定义数据采集的开始时间和间隔。传给 `start` 的时间参数是自纪元（RRDTool 里是指 1970 年 1 月 1 日）起的秒数。`step` 值指明记录保存指标的时间间隔，单位是秒。我们要每分钟保存一次 CPU 利用率，所以 `step` 值设为 60（秒）。

`step` 参数后面一行定义要捕捉的指标。`DS:cpu:GAUGE:120:0:100` 的格式如下：

`DS:variable_name:data_source_type:heartbeat:min:max`

`DS` 关键字表示数据源（Data Source），就是我说的指标。`variable_name` 标识数据源。在上面的例子里，`cpu` 是保存 CPU 利用率的变量名，`data_source_type` 定义值的类型，在本例中是 `GAUGE`。`data_source_type` 可能的取值如下：

- **`COUNTER`**。记录一段时间的变化频率。这种情况下值只会增加。
- **`DERIVE`**。类似 `COUNTER`，但接受负值。
- **`ABSOLUTE`**。也记录变化频率，但当前值总和上个值有关，不同于上个值。用数学术语讲，它总是“delta”。
- **`GAUGE`**。记录实际值，而非变化频率。

RRDTool 按定义好的时间间隔记录值。上面例子里，`myrrddb.rrd` 期望每 60 秒能得到一个 CPU 利用率。RRDTool 数据库期望数据按预定义的时间间隔提供，如果到时候没数据，就记录 `UNDEFINED`，这点和 RDBMS 不同。心跳值（在这个例子里是 120 秒）是数据库认为没数据时记下 `UNDEFINED` 的时间间隔。虽然没按定义的间隔提供数据，但只要在心跳间隔内，RRDTool 就能正确地插入数据。最后两个值 `min` 和 `max` 是边界条件，超出边界的数据会记成 `UNDEFINED`。这个例子里，假设 CPU 利用率是百分比，那么边界值就是 `0` 和 `100`。

最后两行描述聚合函数，它们用于时间序列数据。上面例子最后两行如下：

```
RRA:AVERAGE:0.5:60:24 \
RRA:AVERAGE:0.5:1440:31
```

与 `DS` 类似，`RRA` 也是关键字，表示循环存档（Round Robin Archive）。RRA 定义的格式如下：

```
RRA:consolidation_function:xff:step:rows
```

`consolidation_function` 表示聚合函数，取值可以是 `AVERAGE`、`MINIMUM`、`MAXIMUM` 或 `LAST`。上面有两个 `RRA` 定义，都是求平均。聚合函数输入来自数据，所以这里的 `RRA` 值是对每分钟 CPU 利用率聚合的结果。`step` 定义要聚合多少数据，`rows` 定义要保存多少聚合结果。上面例子里，`step` 值为 60，即每 60 条记录计算一个平均值。CPU 利用率是一分钟一个，所以平均值是每小时一个。归档行数是 24。因此第一个 `RRA` 记录每小时的平均 CPU 利用率，平均值保存一天。

第二个 `RRA` 定义日平均 CPU 利用率，保存 31 天（1 个月）的数据。

RRDTool 能将它记录下的时间序列数据用图形方式展示。可以用命令行脚本或者某种流行的脚本语言来操作数据库。更多有关 RRDTool 的功能及配置的内容请参阅：www.mrtg.org/rrdtool/。

RRDTool 适于监测 NoSQL 集群节点的健康状况。Hypertable 的监控界面就利用了 RRDTool。更多有关 Hypertable 监控界面的内容请参阅：http://code.google.com/p/hypertable/wiki/HypertableManual#Appendix*A*.Monitoring_UI。

17.2　Nagios

Nagios 是开源的宿主与服务监控软件。它利用插件架构提供了极其灵活和可扩展的监控基础设施。Nagios 核心是监控进程，它能监控任意类型的宿主或服务。核心进程完全意识不到它监控的是什么，或者捕捉的指标有什么含义。插件框架建立在核心进程上。插件可以是编译好的可执行文件或脚本（Perl 脚本或命令行脚本）。插件包含连接服务、监控实体以及测量该实体中某个特性的核心逻辑。

插件监测实体并返回结果给 Nagios。Nagios 处理结果并执行动作，比如执行事件处理句柄，或者发送提醒。提醒和警告机制很重要，它们能方便及时沟通。

图 17-1 描述了 Nagios 的架构

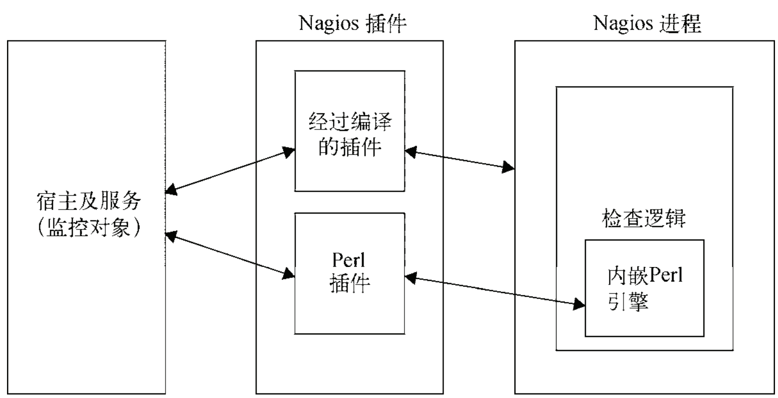


图　17-1

Nagios 可以有效监控 NoSQL 数据库和 Hadoop 集群。Hadoop 和 MongoDB 的插件也已经出

现了一些。由于 Membase 兼容 Memcached，所以 Nagios 也可以监控 Membase，此外还可以添加其他数据库插件。更多有关编写插件的内容请参阅：http://nagios.sourceforge.net/docs/nagioscore/3/en/pluginapi.html。

Nagios 有一个 GPL 许可的 HDFS 插件，地址是：www.matejunkie.com/hadoop-dfs-check-plugin-for-nagios/。监控 MongoDB 的 Nagios 插件的地址是：https://github.com/mzupan/nagios-plugin-mongodb。

Nagios 有很多健壮的插件，它们可以用来监控 CPU 负载、磁盘健康状况、内存使用率和 ping 通率。它们可以监控大多数协议，包括 HTTP、POP3、IMAP、DHCP 和 SSH，也能监控 Linux、Windows 和 Mac OS X 等大多数操作系统上服务的健康状况。更多有关 Nagios 的内容请访问：www.nagios.org/。

17.3　Scribe

Scribe 是开源的、实时的、分布式日志聚合器。它由 Facebook 创建，随后贡献给开源社区。Scribe 非常健壮，是有容错能力的系统。下载地址是：https://github.com/facebook/scribe。Scribe 是分布式系统，集群每个节点都运行本地 Scribe 服务器，其中有一个运行 Scribe 中心（主）服务器。日志先汇集到 Scribe 本地服务器，再发送给中心服务器。如果中心服务器宕机了，日志就写入本地文件，中心服务器重新启动后再发送给它。为避免中心服务器刚启动时负载过大，同步会等到中心服务器启动后，延迟一段时间再发起。

Scribe 日志消息和格式都可配置。它是用非阻塞 C++服务器实现的 Thrift 服务。

Scribe 写日志的可配置性很好。消息映射到类别上，类别映射到特定的存储类型上，存储本身可以有层次结构。不同存储类型如下。

- **文件**：本地文件或 NFS。
- **网络**：发送给另一个 Scribe 服务器。
- **缓冲**（Buffer）：包含主副两个存储。消息发送给主存储，如果主存储宕机，消息就发送给副存储。主存储重新恢复后，消息会再发送给主存储。
- **桶**（Bucket）：包含大量的存储，并形成层次结构。根据哈希决定消息发送给哪个存储。
- **Null**：抛弃所有消息。
- **Thriftfile**：将消息写入一个 Thrift TFileTransport 文件。
- **Multi**：转发器，将消息转发给多个存储。

Scribe 的 Thrift 接口如下：

```
enum ResultCode
{
  OK,
  TRY_LATER
}

struct LogEntry
{
```

```
  1:  string category,
  2:  string message
}
service scribe extends fb303.FacebookService
{
  ResultCode Log(1: list<LogEntry> messages);
}
```

下面是一个简单的 PHP 客户端消息的例子：

```
$messages = array();
$entry = new LogEntry;
$entry->category = "test_bucket";
$entry->message = "a message";
$messages []= $entry;
$result = $conn->Log($messages);
```

scribe_client.php

日志解析与管理是大数据及其处理中非常重要的一项工作。Flume 是继 Scribe 之后的又一解决方案。

17.4 Flume

Flume 是分布式服务，用于高效地收集、聚合和挪动大量日志数据。它基于数据流处理。Flume 非常健壮，有一定的容错能力，支持灵活的配置，文档地址：http://archive.cloudera.com/cdh/3/flume/。

Flume 由多个逻辑节点组成，日志数据流过这些节点。节点可以分成三个独立的层次：

- **代理层**：代理层通常在节点上产生日志文件。
- **收集层**：收集层聚合日志数据，并转与存储层。
- **存储层**：可以是 HDFS。

代理层可以监听来自多个层或数据源的日志。Flume 代理可以监听 syslog 的日志文件、Web 服务器日志或者 Hadoop JobTracker。

Flume 可以看作是由逻辑节点组成的网络，日志数据从源流向最终的存储，每个逻辑节点都定义了源和出口。逻辑节点也可以有修饰器（decorator），但不是必须的。逻辑节点架构支持对数据流进行压缩或批量处理。每个物理节点是一个单独的 Java 进程，多个逻辑节点可以映射到一个物理节点上。

17.5 Chukwa

Chukwa 是 Hadoop 子项目，致力于大规模收集与分析。Chukwa 利用 HDFS 和 MapReduce 提供可扩展的基础设施来聚合和分析日志文件。

与 Scribe 和 Flume 不同，Chukws 还监控和分析工具，而不只是收集和聚合日志。就收集和聚合而言，它很像 Flume。

Chukwa 的架构如图 17-2 所示。

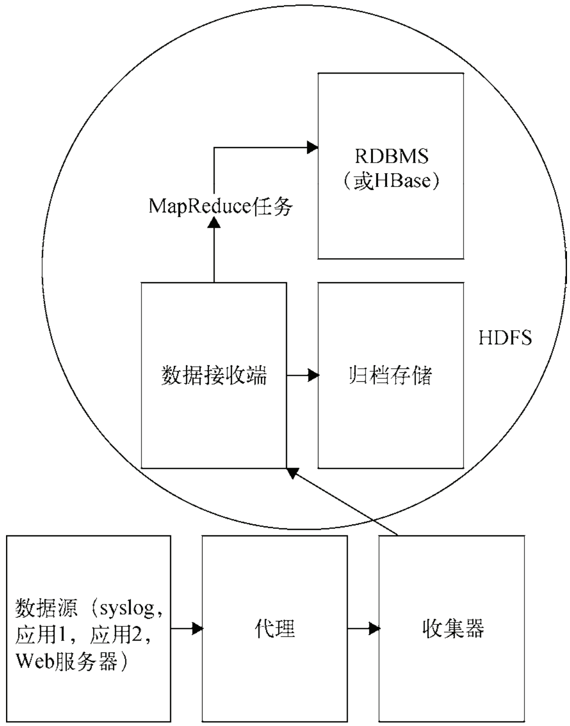


图　17-2

Chukwa 对 Hadoop 的依赖有好处也有坏处。从目前的结构来说，它主要还是批处理工具，不能用于实时分析。

更多有关 Chukwa 的内容可以参阅下面的演讲和研究论文：

- "Chukwa: a scalable log collector"：www.usenix.org/event/lisa10/tech/slides/rabkin.pdf
- "Chukwa: A large-scale monitoringsystem"：www.cca08.org/papers/Paper-13-Ariel-Rabkin.pdf

17.6　Pig

Pig 提供高级数据流定义语言和环境来用 MapReduce 进行大规模数据分析。Pig 包括一种语言 Pig Latin，它的语法简单直观，便于编写并行程序。Pig 通过 MapReduce 任务来有效地执行并行任务。

MapReduce 框架要求开发者从 map 和 reduce 函数的角度来思考，把每个操作都分解成非常简单的步骤，即 map、reduce 两步。map 生成键/值对，reduce 操作或聚合这些键/值对。这种实践要求所有连接、分组、求平均和计数操作都要用 MapReduce 等价方式来定义，这样就限制了开发者的生产力。就 Hadoop 而言，还意味着要编写许多 Java 代码。Pig 提供了高阶抽象和一组

函数。有了 Pig，你不再需要从头开始编写连接、分组、求平均和计数的 MapReduce 任务。此外代码行通常也从数百行 Java 代码缩减成了数十行 Pig Latin 脚本。

Pig 不仅能减少代码行数，紧凑易学的语法也使非程序员执行 MapReduce 任务成为可能。随着 Pig 不断改进，数据科学家和分析师们无需直接使用编程语言就能执行并行任务了。

Pig 提供有语言和执行引擎。执行引擎不仅会翻译和发送任务给 MapReduce 基础设施，同时还管理 Hadoop 配置。大多数时候这些配置都是优化的，说明 Pig 同时还接管了优化配置的责任。这样既获得了额外的优化，也无需更多付出。优化包括选择正确的 reducer 数量或者合适的分区等。

17.6.1 使用Pig

Pig 引擎可以通过下列四种机制来访问。

- 通过脚本。
- 使用命令行接口 grunt。
- 通过 Java 接口 `PigServer` 类。
- 通过 Eclipse 插件。

命令可以写成 Pig Latin 脚本，脚本再提交给 Pig 引擎。另外也可以启动 Pig 命令行 grunt，然后用命令行访问 Pig 引擎。

虽然用 Pig 使得你不用再写 Java 程序来执行 Hadoop MapReduce 任务，但是很可能你还需要用 Java 程序访问 Pig。这种情况下就可以用 Pig 的 Java 类库。类型 `PigServer` 支持 Java 程序通过 JDBC 接口访问 Pig 引擎。在 Java 程序中调用 Pig 时，用 Java 类库而不是外部脚本/程序能减少复杂性。

最后的重要一点，Pig 团队开发了一个叫做 PigPen 的 Eclipse 插件，它提供了强大的 IDE。这个 Eclipse 插件支持图形化地定义数据流，还有脚本开发环境。

17.6.2 Pig Latin基础

Pig Latin 支持下列数据类型：

- Int
- Long
- Double
- Chararray
- Bytearray
- Map（键/值对）
- Tuple（有序列表）
- Bag（无序集）

学习 Pig Latin 以及如何执行 Pig 脚本的最好方法就是看些范例。Pig 自带的范例（包括完整的数据和脚本）位于 tutorial 目录下，那应该算是最好的入门范例。

Pig 教程的 tutorial 目录如下：

- build.xml：ANT 构建脚本。
- data：Excite 搜索引擎日志文件的样例数据。
- scripts：Pig 脚本。
- src：Java 源代码。

介绍所有 Pig 命令语法超出了本书的范围，不过我会介绍一部分范例脚本，好让你对 Pig 有些印象。

tutorial/scripts 目录下有四个脚本，分别为：

- script1-hadoop.pig
- script1-local.pig
- script2-hadoop.pig
- script2-local.pig

-local 脚本在本地执行任务，-hadoop 脚本在 Hadoop 集群上执行任务。这个例子是处理来自搜索引擎 Excite 日志文件的样例数据。脚本 script1-local.pig 用来找出一天中任意时刻的高频搜索词。脚本前面首先注册变量和加载数据，数据经过处理计算出 n-gram 频度。示例脚本的片段如下：

```
-- 调用 NGramGenerator UDF 得到查询语句的 n-gram
ngramed1 = FOREACH houred GENERATE user, hour,
  flatten(
  org.apache.pig.tutorial.NGramGenerator(query))
  as ngram;

-- 用 DISTINCT 获取所有记录中不同 n-gram
ngramed2 = DISTINCT ngramed1;

-- 用 GROUP 按 n-gram 和小时分组
hour_frequency1 = GROUP ngramed2 BY (ngram, hour);

-- 用 COUNT 获取每个 n-gram 出现的次数
hour_frequency2 = FOREACH hour_frequency1 GENERATE flatten($0), COUNT($1) as count;
```

片段展示出了部分 Pig 脚本行。第一行 FOREACH 函数遍历数据生成 n-gram。第二行 DISTINCT 找出唯一的 n-gram。第三行 GROUP 函数按小时归组数据。最后一行遍历分组数据计算 n-gram 出现的频度。这段脚本中 FOREACH 函数用来遍历数据集。很明显 FOREACH、DISTINCT、GROUP 和 COUNT 这样的高级函数简化了数据处理，不需要再用 MapReduce 函数。

Pig 和直接写 MapReduce 相比有一定开销，大约是 1.2 倍，相对于它提供的舒适便利来说还是可以接受的。PigMix（http://wiki.apache.org/pig/PigMix）是一个性能基准工具，用来比较通过 Pig 和直接用 MapReduce 执行任务的性能差别。

在 Yahoo!，Pig 和 Hadoop streaming 是访问 Hadoop 集群的首选方式。Yahoo! 的 Hadoop 集群是世界上最大的 Hadoop 集群之一，它利用这个集群来服务一些关键功能。Pig 在 Yahoo!的使用说明 Pig 已经为在生产环境中使用做好了准备。

更多有关 Pig 的内容可以访问：http://pig.apache.org/。

17.7 Nodetool

Apache Cassandra 是流行的最终一致数据存储。它的分布式特性和最终一致的复制模型使得在运行时可能会出现一些复杂情况。如果能有些工具用来管理和监控 Cassandra 集群就很方便了。nodetool 就是这样一个工具，nodetool 可以像这样运行：

```
bin/nodetool
```

不带参数运行会打印出最常用的命令行参数。Cassandra 节点组成一个环，节点包含的数据映射到一组有序令牌上。所有键都用 MD5 哈希成令牌。要获取 Cassandra 环的状态，可以执行下面的命令：

```
bin/nodetool -host <host_name or ip address> ring
```

其中 `host_name` 或 `ip address` 可以是环上任意的节点。命令输出包括了环上所有节点的状态。它会输出状态、负载、区间和 ACSII 码图。

要获取某个节点的状态，可以执行下面的命令：

```
bin/nodetool -host <host_name or ip address> infc
```

命令输出包含下列内容。

- Token（令牌）
- Load info：磁盘存储的字节数
- Generation no：节点启动次数
- Uptime in seconds（在线时间，单位：秒）
- Heap memory usage（堆的内存使用情况）

Nodetool 还支持一系列其他命令。

- **`ring`**：列出环的信息
- **`info`**：列出节点信息（运行时间、负载等）
- **`cfstats`**：输出列族的统计信息
- **`clearsnapshot`**：移除所有快照
- **`version`**：列出 Cassandra 版本
- **`tpstats`**：输出线程池使用情况
- **`drain`**：排干节点（停止接收写请求，刷新所有列族）
- **`decommission`**：停用节点
- **`loadbalance`**：使节点负载平衡
- **`compactionstats`**：列出压缩的统计信息
- **`disablegossip`**：禁用 gossip（等价于标记节点死亡）
- **`enablegossip`**：重新启用 gossip
- **`disablethrift`**：禁用 Thrift 服务器
- **`enablethrift`**：重启启用 Thrift 服务器
- **`snapshot [snapshotname]`**：创建快照，接受可选的快照名
- **`netstats [host]`**：输出给定主机的网络信息（默认连接节点）

- **move**：将节点移动到新的令牌上
- **removetoken status|force|<token>**：显示删除令牌目前的状态、强制完成删除或删除令牌

更多有关 Nodetool 的内容可以访问：http://wiki.apache.org/cassandra/NodeTool。

17.8　OpenTSDB

随着数据持续增长，你需要向基础设施中添加更多的存储和计算节点，很快集群中就会出现一大批主机、服务器和应用程序，需要管理。大部分主机、服务器和应用程序都提供了钩子以便监控。你可以 ping 这些实体，测量它们的运行时间、性能、使用情况和其他特征。捕获（特别是非常频繁地捕获）、收集和分析这些指标是非常复杂的任务。

OpenTSDB 是分布式可扩展的时间序列数据存储，能非常灵活地管理和监控大量主机、服务器和应用程序。它异步地从大量机器上收集数据并进行存储和索引。OpenTSDB 是开源工具，由 StumbleUpon 团队创建。它用 HBase 存储收集到的数据，支持实时绘图与分析。

OpenTSDB 架构如图 17-3 所示。

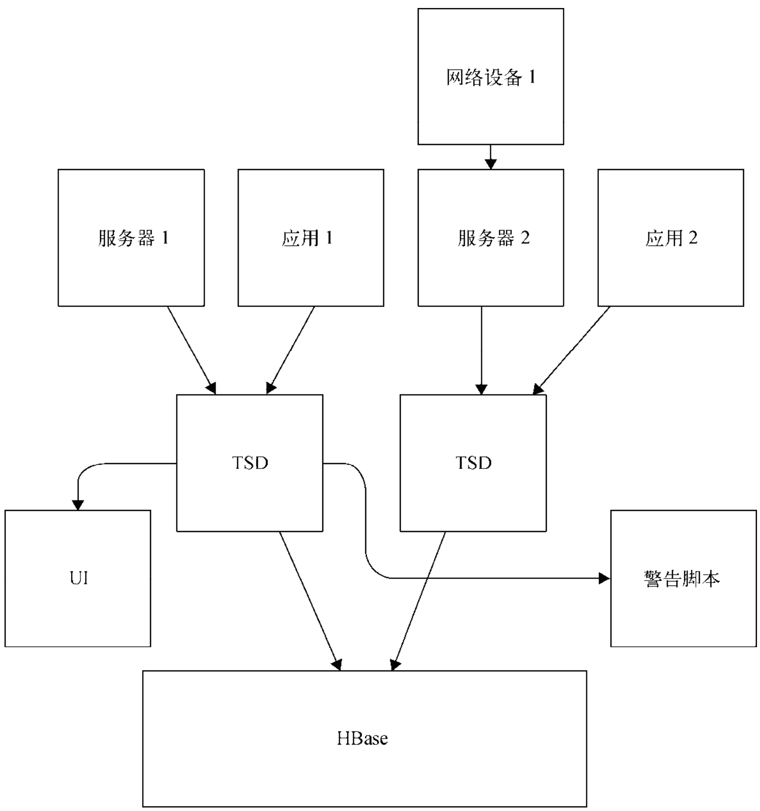


图　17-3

OpenTSDB 能存储数十亿记录，所以不用担心是否需要删除指标和日志数据。在大数据集上做分析能够揭示出一些相关的指标，帮助深入洞察运行系统。OpenTSDB 是分布式的，没有单点故障问题。

更多有关 OpenTSDB 的内容可以访问：http://opentsdb.net/index.html。

17.9 SOLANDRA

Lucene 是流行的开源搜索引擎，Java 编写，过去几年里在许多产品和组织中都得到了应用。Solr 是对 Lucene 类库的封装。Solr 在 Lucene 上提供了 HTTP 服务器、JSON、XML/HTTP 支持等一系列增值功能。而 Solr 底层所有搜索功能都由 Lucene 提供。更多有关 Lucene 的内容可以访问：http://lucene.apache.org/java/docs/index.html，有关 Solr 的内容可以访问：http://lucene.apache.org/solr/。

Solandra 是 Jake Luciani 非常有趣的实验项目。Solandra 项目原本出自 Lucandra，这个项目集成 Lucene 和 Cassandra，并用 Cassandra 存储 Lucene 的索引与文档数据。后来项目转为支持 Solr。

Lucene 是简单优雅的搜索类库，非常容易集成到应用中。核心部分负责管理索引，文档被解析和索引后放到存储中，存储可以是文件系统、内存或任何其他存储系统。查询会被 Lucene 解析并翻译成索引查找。索引读取器读取索引并构造响应，最后返回给调用方。

Solandara 选择 Cassandra 作为存储系统，实现了 Lucene 的 IndexWriter 和 IndexReader 接口，支持将索引和文件写入 Cassandra。图 17-4 和图 17-5 描述了 Solr 和 Solandra 中的索引读写架构。

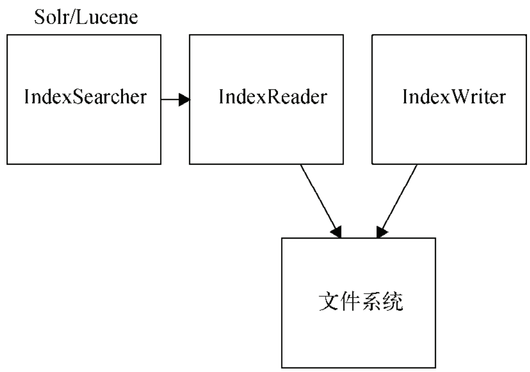


图 17-4

Solandra 定义了两个列族来存储索引和文档。搜索词（search term）列族用来存储索引，键的形式是 `indexName/field/term`，对应的值是`{ documentId , positionVector }`。文档存在另一个列族中，其键形如 `indexName/documentId`，值则是`{ fieldName , value }`。

Solandra 在同一个节点的同一 JVM 中运行 Solr 和 Cassandra。Solandara 索引读写性能比普通的 Solr 要差，不过 Solandra 扩展更容易。如果你已经在用 Cassandra，或者在扩展 Solr 方面遇到

了问题，可以试试 Solandra。Solandra 项目地址：https://github.com/tjake/Solandra。

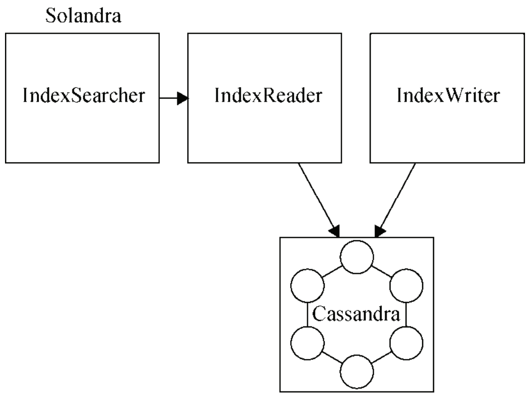


图　17-5

和 Solandra 类似的还有用 HBase 替代 Cassandra 作底层存储，lucehbase 就是这样一个项目，地址：https://github.com/thkoch2001/lucehbase。

如果你想扩展 Lucene，但不是 Cassandra 用户。我不建议用 Solandra，而建议用 Katta。Katta（http://katta.sourceforge.net/）将 Lucene 索引存储到 HDFS 中，从而提供可扩展的分布式搜索软件，它还让你可以有机会利用 Hadoop 基础设施。

17.10　Hummingbird 和 C5T

Hummingbird 是非常活跃的、利用 MongoDB 开发的实时 Web 流量可视化软件，它目前还处于早期阶段，但是非常有趣，令人印象非常深刻，因而值得了解和关注。

Hummingbird 用 node.js 开发，通过 web socket 将数据推送给浏览器。回退方案是用 Flash socket 将数据发送给浏览器。这个项目是开源的，MIT 授权，地址：https://github.com/mnutt/hummingbird。

> Node.js 是 Linux 和 Unix 平台上使用 V8 JavaScript 引擎的事件驱动 I/O 框架，主要用于编写可伸缩的网络程序，例如 Web 服务器。它的设计受到了 Ruby Event Machine 和 Python Twisted 的影响，并和它们类似。更多有关 node.js 的内容请参阅：http://nodejs.org/。

Hummingbird 的实时 Web 流量数据存储在 MongoDB 中，后者提供了快速读写能力。基于 node.js 的跟踪服务器会记录用户在网站上的活动，并将数据存储到 MongoDB 服务器上。已经实现的指标包括到访次数、位置、销售和总浏览数。下面是到访次数的例子：

```
var HitsMetric = {
  name: 'Individual Hits',
  initialData: [],
```

```
    interval: 200,
    incrementCallback: function(view) {
      var value = {
        url: view.env.u,
        timestamp: view.env.timestamp,
        ip: view.env.ip
      };
      this.data.push(value);
    }
  }

  for (var i in HitsMetric)
    exports[i] = HitsMetric[i];
```

更多有关 Hummingbird 的内容可以访问：http://projects.nuttnet.net/hummingbird/。

C5t 是另一个用 MongoDB 构建的有趣软件。它是内容管理软件，用 TurboGears（Python Web 框架）和 MongoDB 写成。源代码地址：https://bitbucket.org/percious/c5t/wiki/Home。

只要输入你想要的 URL 就能创建页面，页面可以是公共的或者私有的，此外它还提供了内置的认证授权和全文搜索功能。

17.11 GeoCouch

GeoCouch 是 CouchDB 的扩展，为 Apache CouchDB 提供地理信息索引。项目地址：https://github.com/couchbase/geocouch，同时它还被收入 Couchbase 公司（赞助了该项目的开发）的 Couchbase 中。GeoCouch 第一版用 Python 和 SpatiaLite 开发，通过 stdin 和 stdout 与 CouchDB 交互。目前版本用 Erlang 开发，和 CouchDB 的集成也更好。

地理信息索引引入了有关数据位置的视角。随着 GPS、位置感应器、地图和本地搜索的出现，地理信息索引正逐渐成为许多应用中的重要部分。

GeoCouch 支持许多种地理信息索引类型，如下所示：

- Point
- Polygon
- LineString
- MultiPoint
- MultiPolygon
- MultiLineString
- GeometryCollection

GeoCouch 用 R 树存储地理信息索引。R 树（http://en.wikipedia.org/wiki/R-tree）广泛用于多种地理信息产品，比如 PostGIS、SptiaLite 和 Oracle Spatial。R 树使用方块作为地理位置的近似值，对展示大部分几何图形来说足够了。

PDX API（www.pdxapi.com/）是一个很好的 GeoCouch 学习范例，它利用 GeoCouch 为波兰市的开放地理数据创建了一个 REST 服务。波兰将其地理数据以 shapefile 形式公布。这些文件通过 PostGIS 转换成 GeoJSON。CouchDB 支持 JSON，能很容易导入 GeoJSON 并轻松提供强大的 REST API。

17.12　Alchemy Database

Alchemy 数据库（http://code.google.com/p/alchemydatabase/）是具有双重性质的数据库。它既可以是 RDBMS，又可以是 NoSQL。它构建在 Redis 和 Lua 上，内嵌 Lua 解释器。由于使用 Redis，数据库速度非常快，而且还尽量在内存中完成各种操作。不过这也意味着，如果内存数据和磁盘上整个数据集差别很大时，它会受到 Redis 的限制。

Alchemy 的性能惊人，原因如下：

- 使用事件驱动的网络服务器，尽可能利用内存。
- 高效数据结构和压缩允许在 RAM 中存放很多数据。
- 只支持 OLTP 最相关的 SQL 语句，保持系统轻量级，同时又很有用。不支持复杂 SQL。Alchemy 能支持的 SQL 命令列表地址：http://code.google.com/p/alchemydatabase/wiki/CommandReference#Supported_SQL。

17.13　Webdis

Webdis（http://webd.is）是 Redis 的高速 HTTP 接口。它就是一个简单的 Web 服务器，请求下发给 Redis，响应返回给客户端。Webdis 默认支持 JSON，此外还支持其他类型如下。

- 文本， text/plain。
- HTML、XML、PNG、JPEG、PDF，分别使用各自的扩展类型。
- BSON，application/bson。
- Redis 协议格式。

Webdis 行为类似普通的 Web 服务器，只不过略有修改以支持 Redis 命令。普通请求返回 200 ok 代码。如果受访问控制影响，不允许回应某个请求，客户端会收到 403 forbidden 的 HTTP 响应。不支持 GET、POST 和 OPTIONS，使用它们会返回 405 Method Not Allowed。Webdis 支持 HTTP PUT，可以通过命令来设置值，例如像下面这样：

```
curl --upload-file my-data.bin http://127.0.0.1:7379/SET/akey
```

17.14　小结

在我写本章的总结时，期望你能从学习这些重要技术的过程中得到丰富有趣的体验。本章展示 NoSQL 相关工具和实用程序，及相关案例。

像 RRDTool 和 Nagios 这样的工具是通用的，都是非常有价值的监控及管理软件。像 nodetool 这样的工具在管理和监控 Cassandra 时，能带来额外的好处。

Scribe、Flume 和 Chukwa 在分布式日志处理和聚合方面提供了非常强大的能力。它们提供了健壮的功能来管理分布式环境中生成的大量日志文件。OpenTSDB 为主机、服务和应用程序提供了实时监控的基础设施。

Pig 是非常有价值的工具，可用于在 Hadoop 集群上编写聪明的 MapReduce 任务，本章帮你在这方面开了个头。像 Solandra、Hummingbird、c5t、GeoCouch、Alchemy 数据库和 Webdis 这样有趣的应用能让我们看到，当 NoSQL 产品的灵活强大和你的点子碰撞在一起时，会激发出怎样的火花。本章所罗列的各种工具、实用程序和用例不过是大千世界的小小一角。读过本书的学习，希望能激发起你的兴趣，去了解和学习更有趣，更适合你的 NoSQL 产品。

附录A

安装与配置

本章内容

- ❑ 安装并设置几个流行的 NoSQL 产品
- ❑ 了解不同平台上基本安装的区别
- ❑ 源码编译适用的产品
- ❑ 配置 NoSQL 产品

软件的安装说明常因操作系统不同而变化。这里包含的大部分说明对 Linux、Unix 和 Mac OS X 都有效，有些地方也会介绍在 Windows 上的安装说明。

我们经常要在`/opt` 目录下安装组件。root 以外用户默认无权限写此目录。如果文中指定说要在`/opt` 目录下解压文件或执行其他操作，用户又没有写权限，可以运行加上 `sudo(8)`下的命令或者使用 `chmod(1)`把`/opt` 目录改成可写的。

A.1 Hadoop安装与配置

本节说明如何安装配置 Hadoop Common、HDFS 以及 Hadoop MapReduce。

以下软件是必需的。

- ❑ **Java 1.6.x**。Hadoop 是在 Sun（现在是 Oracle）JDK 下进行的测试。
- ❑ **SSH 和 sshd**。SSH 一定要装，sshd 也得跑起来。Hadoop 脚本通过 sshd 连接远程 Hadoop 进程。

Hadoop 可安装并运行为单节点或多节点集群。单节点模式也可配置成伪分布式。本节主要关注单节点和伪分布式模式，不包括集群，不过会提供这个主题的文档链接。

A.1.1 安装

(1) 下载 Hadoop 稳定版，地址：http://hadoop.apache.org/common/releases.html。编写本书时 Hadoop 最新版本是 0.21.0，我们选 0.20.2。用这个版本能避免 0.21.0 造成的不一致问题，特别是和 HBase 一起用时。

(2) 解压文件。

(3) 解压完找个地方放好，我选择放到`/opt` 目录里。

(4) （可选）在执行解压的目录里创建符号链接 hadoop 指向上面步骤的目录。符号链接可以像下面这样创建：ln -s hadoop-0.20.2 hadoop，此命令假设你正处在提取文档的目录下。

安装完 Hadoop 后，执行下列基本配置步骤。

(1) 修改 conf/hadoop-env.sh，设置 JAVA_HOME 为相应的 JDK。Ubuntu+OpenJDK 的话，JAVA_HOME 应该是/usr/lib/jvm/java-1.6.0-openjdk。Mac OS X 上 JAVA_HOME 多半是/System/Library/Frameworks/JavaVM.framework/Versions/1.6.0/Home。

(2) 执行 bin/hadoop，如果看到下面的输出就说明安装没问题：

```
Usage: hadoop [--config confdir] COMMAND
where COMMAND is one of:
  namenode -format format the DFS filesystem
  secondarynamenoderun the DFS secondary namenode
  namenode          run the DFS namenode
  datanode          run a DFS datanode
  dfsadmin          run a DFS admin client
  mradmin           run a Map-Reduce admin client
  fsck              run a DFS filesystem checking utility
  fs                run a generic filesystem user client
  balancer          run a cluster balancing utility
  jobtracker        run the MapReduce job Tracker node
  pipes             run a Pipes job
  tasktracker       run a MapReduce task Tracker node
  job               manipulate MapReduce jobs
  queue             get information regarding JobQueues
  version           print the version
  jar <jar>         run a jar file
  distcp <srcurl> <desturl> copy file or directories recursively
  archive -archiveName NAME <src>* <dest> create a hadoop archive
  daemonlog         get/set the log level for each daemon
 or
  CLASSNAME         run the class named CLASSNAME
Most commands print help when invoked w/o parameters.
```

如果没看到 Hadoop 命令行选项，就要检查 JAVA_HOME 是否指到了正确的 JDK 上。

A.1.2　单节点配置

Hadoop默认以单节点模式运行。要试一下Hadoop是否能正常工作，可以像下面这样实验HDFS：

- $ mkdir input
- $ cp bin/*.sh input
- $ bin/hadoop jar hadoop-examples-*.jar grep input output 'start[a-z.]+'

该命令应该会触发 MapReduce 任务，任务生成输出的开始部分是这样的：

```
<date time>INFO jvm.JvmMetrics: Initializing JVM Metrics with
 processName=JobTracker, sessionId=
<date time>INFO mapred.FileInputFormat: Total input paths to process : 12
<date time> INFO mapred.JobClient: Running job: job_local_0001
<date time> INFO mapred.FileInputFormat: Total input paths to process : 12
<date time> INFO mapred.MapTask: numReduceTasks: 1
```

```
<date time> INFO mapred.MapTask: io.sort.mb = 100
<date time> INFO mapred.MapTask: data buffer = 79691776/99614720
<date time> INFO mapred.MapTask: record buffer = 262144/327680
<date time> INFO mapred.MapTask: Starting flush of map output
<date time> INFO mapred.TaskRunner: Task:attempt_local_0001_m_000000_0 is done.
And is in the process of commiting
<date time> INFO mapred.LocalJobRunner: file:/opt/hadoop-0.20.2/input/hadoop-
config.sh:0+1966
...
```

可以用命令**cat output/***确认输出的内容。

输出可能会因 Hadoop 版本有所变化，大概是这样的：

```
2    starting
1    starts
1    startup
```

A.1.3 伪分布式模式配置

要把 Hadoop 配置成伪分布式，一个重要前提是不用口令就能 SSH 到 localhost。

试试：

```
ssh localhost
```

系统会提示你确认本地服务器的可靠性，输入 yes 来响应该提示。

如果不用输入密码就能登录成功，那就可以继续了。否则的话，应该先执行下面的命令设置密钥认证（无需密码）：

```
$ ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Hadoop 可以在单节点上以伪分布式模式运行，每个 Hadoop 守护进程运行在独立的 Java 进程中。

下面是基本安装步骤。

(1) 修改 conf/core-site.xml，用下面的内容来替换空的<configuration></configuration>标签：

```
<configuration>
  <property>
       <name>fs.default.name</name>
       <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

（配置 Hadoop 守护进程）

(2) 修改 conf/hdfs-site.xml，用下面的内容来替换空的<configuration></configuration>标签：

```
<configuration>
  <property>
       <name>dfs.replication</name>
       <value>1</value>
  </property>
</configuration>
```

（配置复制因子，1 表明无复制。更高的复制因子需要更多的节点）

(3) 修改 conf/mapred-site.xml，用下面内容来替换空的<configuration></configuration>标签：

```
<configuration>
  <property>
       <name>mapred.job.tracker</name>
       <value>localhost:9001</value>
  </property>
</configuration>
```

（配置 MapReduce 守护进程）

(4) 下面通过格式化 HDFS 来测试伪分布式模式：

```
bin/hadoop namenode -format
```

如果看到类似下面这样的输出（host 不一样），那就说明一切 OK：

```
11/05/26 23:05:36 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = treasuryofideas-desktop/127.0.1.1
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 0.20.2
STARTUP_MSG:   build =
https://svn.apache.org/repos/asf/hadoop/common/branches/branch-0.20 -r 911707;
 compiled by 'chrisdo' on Fri Feb 19 08:07:34 UTC 2010
************************************************************/
11/05/26 23:05:37 INFO namenode.FSNamesystem:
 fsOwner=treasuryofideas,treasuryofideas,adm,dialout,
cdrom,plugdev,lpadmin,admin,sambashare
11/05/26 23:05:37 INFO namenode.FSNamesystem: supergroup=supergroup
11/05/26 23:05:37 INFO namenode.FSNamesystem: isPermissionEnabled=true
11/05/26 23:05:37 INFO common.Storage: Image file of size 105 saved in 0 seconds.
11/05/26 23:05:37 INFO common.Storage: Storage directory /tmp/hadoop-
treasuryofideas/dfs/name has been successfully formatted.
11/05/26 23:05:37 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at treasuryofideas-desktop/127.0.1.1
************************************************************/
```

(5) 启动所有 Hadoop 守护进程：

```
bin/start-all.sh
```

(6) 检查日志文件确认所有组件都运行了，日志默认放在 log 目录下面：

你会看到下列日志文件（以你的用户名代替 username，你的主机名代替 hostname）：

```
$ ls logs/
hadoop-username-datanode-hostname.log
hadoop-username-datanode-hostname.out
hadoop-username-jobtracker-hostname.log
hadoop-username-jobtracker-hostname.out
hadoop-username-namenode-hostname.log
hadoop-username-namenode-hostname.out
```

```
hadoop-username-secondarynamenode-hostname.log
hadoop-username-secondarynamenode-hostname.out
hadoop-username-tasktracker-hostname.log
hadoop-username-tasktracker-hostname.out
history/
```

(7) 访问 Namenode 和 JobTracker 的 Web 接口，地址分别是 http://localhost:50070/和 http://localhost:50030/：

(8) 运行 jps 列出所有 Java 进程。你应该会看到下列输出，当然可能还包括其他正在运行的 Java 进程：

```
2675 JobTracker
2442 DataNode
2279 NameNode
3027 Jps
2828 TaskTracker
2603 SecondaryNameNode
```

（进程 ID 很可能不同）

(9) 接下来重新运行 HDFS 测试驱动：

```
bin/hadoop fs -put bin input
bin/hadoop jar hadoop-*-examples.jar grep input output 'start[a-z.]+'

11/06/04 11:53:07 INFO mapred.FileInputFormat: Total input paths to process : 17
11/06/04 11:53:08 INFO mapred.JobClient: Running job: job_201106041151_0001
11/06/04 11:53:09 INFO mapred.JobClient: map 0% reduce 0%
11/06/04 11:53:24 INFO mapred.JobClient: map 11% reduce 0%
(...)
11/06/04 11:54:58 INFO mapred.JobClient: map 100% reduce 27%
11/06/04 11:55:10 INFO mapred.JobClient: map 100% reduce 100%
11/06/04 11:55:15 INFO mapred.JobClient: Job complete: job_201106041151_0001
(...)
11/06/04 11:55:48 INFO mapred.JobClient: Combine output records=0
11/06/04 11:55:48 INFO mapred.JobClient: Reduce output records=4
11/06/04 11:55:48 INFO mapred.JobClient: Map output records=4
```

(10) 为确认输出，把输出从 HDFS 拷贝到本地文件系统，然后将内容打印到标准输出上：

```
bin/hadoop fs -get output
pseudo-output
cat pseudo-output/*
cat: psuedo-output/_logs: Is a directory
5     starting
1     started
1     starts
1     startup
```

(11) 直接从 HDFS 上输出 MapReduce 任务，应该能匹配本地文件系统上获得的输出：

```
bin/hadoop fs -cat output/part*
5     starting
1     started
1     starts
1     startup
```

这样就完成了伪分布式的设置。有关Hadoop集群的安装配置请参阅：http://hadoop.apache.org/common/docs/r0.20.2/cluster_setup.html。

A.2　HBase安装与配置

HBase 独立模式的安装配置步骤如下。

(1) 下载最新的稳定版 HBase，地址：www.apache.org/dyn/closer.cgi/hbase/。本书编写时最新稳定版是 hbase-0.90.3。要特别注意它和 Hadoop 版本的兼容性，因为 HBase 有一些依赖。

(2) 解压 HBase 文件：

```
tar zxvf hbase-0.90.3.tar.gz.
```

(3) 解压完找个地方放好，我选择/opt 目录。

(4) 创建符号链接 ln -s hbase-0.90.3 hbase。

(5) 修改配置文件 conf/hbase-site.xml，将<configuration.</configuration>替换成：

```
<configuration>
  <property>
       <name>hbase.rootdir</name>
       <value>file:///opt/hbase_rootdir</value>
  </property>
</configuration>
```

hbase.rootdir是HBase写入的目录。我的hbase.rootdir设成了/opt/hbase_rootdir。你可以改成其他位置。hbase.rootdir 默认是/tmp/hbase-${user.name}，服务器重启时有可能被删除。

(6) 启动 HBase 确认安装成功：

```
bin/start-hbase.sh
```

(7) 用 shell 连接 HBase：

```
bin/hbase shell
```

A.3　Hive安装与配置

以下列出必需的软件。

- Java 1.6.x。Hadoop 是在 Sun（现在是 Oracle）JDK 下测试的。
- Hadoop 0.20.2。Hive 能用 0.17.x 和 0.20.x 版本的 Hadoop。

安装配置步骤如下。

(1) 下载稳定版，地址：www.apache.org/dyn/closer.cgi/hive/。编写本书时的稳定版本是 hive-0.70.0。二进制版文件名里有-bin，因为是 Java 编写的所以跨平台。

(2) 解压：

```
tar zxvf hive-0.7.0-bin.tar.gz.
```

(3) 解压完找个地方放好，我选择/opt 目录。

(4) 创建符号连接：

```
ln -s hive-0.7.0-bin hive
```

(5) 设置 HIVE_HOME 环境变量：

```
export HIVE_HOME=/opt/hive
```

（指向的目录中包含 Hive）

(6) 把 Hive 添加到 PATH 里：

```
export PATH=$HIVE_HOME/bin:$PATH
```

(7) 执行 which hive 以确认设置成功，应该能看见配置好的 PATH：

```
which hive
/opt/hive/bin/hive
```

A.3.1　配置

确认 Hadoop 添加到了 PATH 里，或者 HADOOP_HOME 环境变量指向 Hadoop 目录。可以这样设置 HADOOP_HOME：

(1) 设置环境变量：

```
export  HADOOP_HOME=/opt/hadoop
```

（指向包含 Hadoop 的目录）

(2) 在 HDFS 上创建/tmp 目录：

```
$HADOOP_HOME/bin/hadoop fs -mkdir       /tmp
```

（注意目录可能已经存在）

(3) 给 HDFS 上的/tmp 目录增加组用户的写权限：

```
$HADOOP_HOME/bin/hadoop fs -chmod g+w   /tmp\
```

(4) 创建 hive.metastore.warehouse.dir 目录（默认是/user/hive/warehouse）

```
$HADOOP_HOME/bin/hadoop fs -mkdir       /user/hive/warehouse
```

(5) 指定 HDFS 目录/user/hive/warehouse 的写权限：

```
$HADOOP_HOME/bin/hadoop fs -chmod g+w   /user/hive/warehouse
```

A.3.2　Hadoop覆盖配置

Hive 配置建立在 Hadoop 配置基础上。Hive 默认配置包含在 conf/hive-default.xml 中，通过修改这个文件中的变量可以覆盖默认配置，通过修改 HIVE_CONF_DIR 可以修改 Hive 配置目录指向新配置。除了修改 conf/hive-site.xml 的配置外，还可以用以下方法修改配置。

- Hive 命令行 SET 命令。比如 hive > SET mapred.job.tracker=hostName.organiza-

tionName.com:50030;设置 MapReduce 集群配置。

- hiveconf 配置变量。将 hiveconfig 变量值传给 hive 程序。例如 bin/hive -mapred.job.tracker=hostName.organizationName.com:50030 设置 MapReduce 集群和上面的 SET 命令完全一样。hiveconf 一次可以传入多个，不过传入大量参数不太容易维护，可以把所有变量设置为环境变量 HIVE_OPTS 的值。

要确认 Hive 安装成功，需执行$HIVE_HOME/bin/hive。

A.4　Hypertable安装与配置

Hypertable 最简单的安装方式是用二进制包，它和所有用 glibc 2.4+的系统都兼容。如果你的系统使用更老的 glibc，则需要手动编译打包 Hypertable，具体说明请参阅：http://code.google.com/p/hypertable/wiki/HowToPackage。

Hypertable 同时提供 32 位及 64 位平台的二进制包，提供的格式包括.rpm、.deb、.dmg。此外还有.tar.gz2 的。为中立起见，我选择.tar.gz2。

参照下面的步骤安装 64 位 Hypertable .tar.gz2 包。

(1) 下载最新发布版地址：www.hypertable.com/download/。编写本书时的最新版是 0.9.5.0.pre5。

(2) 解压缩：

```
tar jxvf hypertable-0.9.5.0.pre5-linux-x86_64.tar.bz2.
```

(3) 解压完找个地方放好，我选择/opt 目录，结构如下：

```
/opt/hypertable/<version>
```

推荐按这个结构放，命令如下：

```
cd hypertable-0.9.5.0.pre5-linux-x86_64/opt, and
mv hypertable /opt
```

(4) 创建符号链接**current**：

```
ln -s /opt/hypertable/0.9.5.0.pre5 /opt/hypertable/current
```

(5) 也可以使其兼容 FHS（文件系统层次标准，Filesystem Hierarchy Standard），此项可选。

A.4.1　FHS兼容

FHS 是 Linux/Unix 文件系统推荐的文件组织方式。标准建议把软件包的配置放到/etc/opt 中，数据放到/var/opt 中。

要让 Hypertable 兼容 FHS，请参照下列步骤。

(1) 用你的用户和组（可用 id 命令获取到）替换<userName>:<groupName>并执行下面的命令：

```
sudo mkdir /etc/opt/hypertable /var/opt/hypertable
sudo chown <userName>:<groupName> /etc/opt/hypertable /var/opt/hypertable
```

(2) 运行下面的命令：

```
$ bin/fhsize.sh:
Setting up /var/opt/hypertable
Setting up /etc/opt/hypertable
fshize /opt/hypertable/current:  success
```

在升级 Hypertable 时，FHS 兼容能避免重新创建配置、日志、hyperspace 和 localBroker 根目录。

(3) 确认 Hypertable 兼容 FHS，列出`/opt/hypertable/current` 目录内容并确认符号链接，目录应该是像下面这样：

```
$ cd /opt/hypertable/current
$ ls -l
bin
conf -> /etc/opt/hypertable
examples
fs -> /var/opt/hypertable/fs
hyperspace -> /var/opt/hypertable/hyperspace
include
lib
log -> /var/opt/hypertable/log
Monitoring
run -> /var/opt/hypertable/run
```

A.4.2　配置Hadoop和Hypertable

如果你有按本章说明配置 Hadoop，那 `conf/core-site.xml` 里的 HDFS 配置应该是这样：

```
<configuration>
  <property>
       <name>fs.default.name</name>
       <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

（`conf/core-site.xml` 的内容）

现在要修改 `conf/hypertable.cfg`：

(1) 确认 `hypertable.cfg` 里的有关 HDFS 的配置 `HdfsBroker.fs.default.name` 是：

```
# HDFS Broker
HdfsBroker.fs.default.name=hdfs://localhost:9000
matches with the HDFS daemon configuration in Hadoop conf/core-site.xml
Create /hypertable directory on HDFS:
$HADOOP_HOME/bin/hadoop fs -mkdir       /hypertable
```

(2) 修改`/hypertable` 目录的写权限：

```
$HADOOP_HOME/bin/hadoop fs -chmod g+w   /hypertable
```

A.5　MongoDB安装与配置

从 www.mongodb.org/downloads 下载最新版本 MongoDB，其中包含大多主流操作系统的二

进制包。我下载的是 64 位 Linux 的 1.8.2-rc2 版。如果你选择其他版本，大部分安装步骤应该也还是下面这样的。

(1) 解压缩：

```
tar zxvf mongodb-osx-x86_64-1.8.2-rc2.tgz
```

(2) 解压完找个地方放好，我选择/opt 目录，结构如下：

```
mv mongodb-osx-x86_64-1.8.2-rc2 /opt
```

(3) 创建符号链接 mongodb 指向目录：

```
ln -s mongodb-osx-x86_64-1.8.2-rc2 mongodb
```

配置MongoDB

MongoDB 数据默认存放在/data/db 目录下。如果想用默认目录，需创建目录并设好权限：

```
$ sudo mkdir -p /data/db
$ sudo chown `id -u` /data/db
```

如果想用别的目录存放 MongoDB 数据文件，比如/opt/data/db，可以这样做：

```
$ sudo mkdir -p /opt/data/db
$ sudo chown `id -u` /opt/data/db
```

不用默认值的话，记得把新目录作为--dbpath 的参数传给 MongoDB，像这样：

```
bin/mongod --dbpath /opt/data/db
```

A.6　CouchDB安装与配置

要安装 CouchDB，首先需要安装 Erlang 和 Erlang OTP。

Linux 和 Unix 上装 Erlang 挺简单的。Mac OS X 上可以利用 brew（http://mxcl.github.com/homebrew/）安装 Erlang。Windows 上最简单的方式是安装 Couchbase 的 Couchbase Server1.1，地址：www.couchbase.com/downloads，它包括了 Erlang Widnows 版、CouchDB 还有其他一些特性。有关在 Windows 上安装 Apache Couch 的说明可以在这里找到：http://wiki.apache.org/couchdb/Installing_on_Windows，其中也包括 Erlang 的安装步骤。

Apache CouchDB 安装包大多数平台都有，安装包和说明可以在如下地址找到：http://wiki.apache.org/couchdb/Installation。CouchDB 背后的公司 Couchbase 也为许多平台提供了二进制安装包。

前面大部分安装说明都是用于安装二进制文件。本附录罗列的所有软件都开源，源代码自由获取，因此你也可以选择用源代码编译安装。下面作为一个例子，我们演示如何在 Ubuntu 10.04 上编译安装 CouchDB。

Ubuntu10.04 的CouchDB源代码安装

在 Ubuntu Linux 用源代码安装 CouchDB 可以参照下面的步骤。

(1) 安装依赖项：

```
sudo apt-get build-dep couchdb
sudo apt-get install xulrunner-1.9.2-dev libicu-dev libcurl4-gnutls-dev libtool
```

(2) 获取 xulrunner 版本：

```
xulrunner -v
```

我的机器上 Ubuntu 10.04 输出是 `Mozilla XULRunner 1.9.2.17 -20110424212116`。

(3) 创建 xulrunner 共享类库加载配置，这是因为可能有多个 xulrunner 版本：

```
sudo vi /etc/ld.so.conf.d/xulrunner.conf
```

(4) 添加下列行：

```
/usr/lib/xulrunner-1.9.2.17
/usr/lib/xulrunner-devel-1.9.2.17
```

(5) 执行 `ldconfig`：

```
sudo /sbin/ldconfig
```

(6) 获取源码，可以用 SVN 或 Git：

```
git clone git://git.apache.org/couchdb.git
```

(7) 进入源码目录：

```
cd couchdb
```

(8) Bootstrap 之：

```
./bootstrap
```

注意，如果这步出错，你可能需要安装依赖项，`INSTALL.Unix` 文件描述了所有依赖项。另外可能还需要安装 `.aclocal`，命令为：`sudo apt-get install automake`。

(9) 配置：

```
./configure
```

(10) 编译安装：

```
make && sudo make install
```

(11) 创建一个名为 `couchdb` 的用户：

```
useradd couchdb
```

(12) 修改 CouchDB 目录权限给 `couchdb` 用户。

(13) 修改 CouchDB 目录所有权给 `couchdb` 用户。

```
chown -R couchdb:couchdb /usr/local/etc/default/couchdb
chown -R couchdb:couchdb /usr/local/etc/init.d/couchdb
chown -R couchdb:couchdb /usr/local/etc/couchdb
chown -R couchdb:couchdb /usr/local/etc/logrotate.d/couchdb
chown -R couchdb:couchdb /usr/local/lib/couchdb
chown -R couchdb:couchdb /usr/local/bin/couchdb
chown -R couchdb:couchdb /usr/local/var/lib/couchdb
```

```
chown -R couchdb:couchdb /usr/local/var/run/couchdb
chown -R couchdb:couchdb /usr/local/var/log/couchdb
chown -R couchdb:couchdb /usr/local/share/doc/couchdb
chown -R couchdb:couchdb /usr/local/share/couchdb
```

A.7　Redis安装与配置

参照下列步骤安装 Redis。

(1) 下载最新的稳定版本，地址：http://redis.io/download。编写本书时最新版本是 2.2.8。

(2) 解压缩：`tar zxvf redis-2.2.8.tar.gz`

(3) 将文件移动到`/opt` 目录：`mv redis-2.2.8 /opt`

(4) 创建链接：

```
ln -s redis-2.2.8 redis
```

(5) 编译：

```
cd redis
make
```

(6) `make test.`来确认。

A.8　Cassandra安装与配置

参照下列步骤安装 Cassandra。

(1) 下载二进制开发版地址：http://cassandra.apache.org/download/。编写本书时最新版本是 0.8.0-rc1。下载文件为：`apache-cassandra-0.8.0-rc1-bin.tar.gz`。

(2) 解压缩：

```
tar zxvf apache-cassandra-0.8.0-rc1-bin.tar.gz
```

(3) 将文件移动到目标目录：

```
mv apache-cassandra-0.8.0-rc1 /opt
```

(4) 创建名为 `apache-cassandra` 的链接指向包含 Cassandra 的目录：

```
ln -s apache-cassandra-0.8.0-rc1 apache-cassandra
```

A.8.1　配置Cassandra

Cassandra 通过文件 `conf/cassandra.yaml` 配置。大部分默认配置适用于单节点模式，只要确认 `cassandra.yaml` 中声明的目录都存在就行。

下列配置指向文件系统：

```
# directories where Cassandra should store data on disk.
data_file_directories:
    - /var/lib/cassandra/data
```

```
# commit log
commitlog_directory: /var/lib/cassandra/commitlog

# saved caches
saved_caches_directory: /var/lib/cassandra/saved_caches
```

用命令 `sudo mkdir -p /var/lib/cassandra` 创建`/var/lib/cassandra`。确保该目录设置了适当的权限，以便运行 Cassandra 进程可以写这个目录。

A.8.2　配置log4j

log4j 服务器属性声明在文件 `log4j-server.properties` 中。Log4j 的 appender 文件声明如下：

```
log4j.appender.R.File=/var/log/cassandra/system.log
```

确保目录`/var/log/cassandra` 存在并设置适当的写权限，以便运行 Cassandra 进程可以写这个目录。

A.8.3　Cassandra源码安装

需要下列软件：

- Java 1.6.x
- Ant 1.8.2

参照下面的步骤编译 Cassandra 源码。

(1) 从 http://cassandra.apache.org/download/下载最新的开发版源码。编写本书时的版本是 0.8.0.rc1。

(2) 解压缩：

```
tar zxvf apache-cassandra-0.8.0-rc1-src.tar.gz.
```

(3) 执行 Ant 编译任务：

```
ant
```

A.9　Membase Server和Memcached安装与配置

从 www.couchbase.com/downloads 下载相关版本。有如下三个不同的版本可以下载安装：

- Membase 服务器
- Memcached 服务器
- Couchbase 服务器

从上面网址下载适合你的操作系统的 Membase。二进制版容易安装。下面用 Mac OS X 演示安装。

以下步骤均与 Mac OS X 相关。

(1) Mac OS X 版 Membase 打成 zip 包，文件是：`membase-server-community-1.6.5.3.zip`

(2) 解压缩：

```
unzip membase-server-community-1.6.5.3.zip.
```

解压缩完的程序位于 Mac OS X 格式的目录 `Membase.app` 中。

(3) 将 `Membase.app` 移动到 `/Application` 目录或其它保存应用程序的目录中。

A.10　Nagios安装与配置

本节只介绍在 Ubuntu 上源码安装 Nagios。更多细节请参阅 Nagios 文档，地址：www.nagios.org/documentation。

下列软件是必需的：

- Apache 2
- PHP
- GCC（http://gcc.gnu.org/）、编译器及开发包
- GD 开发包

参照下面的步骤安装必需软件。

(1) 安装 Apache 2：

```
sudo apt-get install apache2
```

(2) 安装 PHP：

```
sudo apt-get install libapache2-mod-php5
```

(3) 安装 GCC 和开发包：

```
sudo apt-get install build-essential
```

(4) 安装 GD 开发包：

```
sudo apt-get install libgd2-xpm-dev
```

推荐创建一个名为 `nagios` 的用户来运行 Nagios 进程。在 Ubuntu 上创建 `nagions` 用户如下：

```
sudo /usr/sbin/useradd -m -s /bin/bash nagios
sudo passwd nagios
```

（设个密码，我一般用 `nagios`。命令行会提示输入密码并再次确认。）

创建 `nagcmd` 组，然后把 `nagios` 和 `apache` 用户都添加到这个组里：

```
sudo /usr/sbin/groupadd nagcmd
sudo /usr/sbin/usermod -a -G nagcmd nagios
sudo sudo /usr/sbin/usermod -a -G nagcmd nagios
```

A.10.1　下载和编译Nagios

安装完所有必需软件后，按如下步骤下载并安装 Nagios：

(1) 下载 Nagios Core 和 Nagios Plugins，地址：www.nagios.org/download/。编写本书时 Nagios Core 版本是 3.2.3，Nagios Plugins 版本是 1.4.15。

(2) 解压缩：

```
tar zxvf nagios-3.2.3.tar.gz
```

(3) 进入 nagios-3.2.3 目录：

```
cd nagios-3.2.3
```

(4) 配置 Nagios：

```
./configure --with-command-group=nagcmd
```

(5) 编译：

```
make all
```

(6) 安装：

```
sudo make install
```

(7) 安装 init 脚本：

```
sudo make install-init
```

命令输入如下：

```
/usr/bin/install -c -m 755 -d -o root -g root /etc/init.d
/usr/bin/install -c -m 755 -o root -g root daemon-init /etc/init.d/nagios

*** Init script installed ***
```

(8) 安装样例配置文件：

```
sudo make install-config.
```

输出如下：

```
/usr/bin/install -c -m 775 -o nagios -g nagios -d /usr/local/nagios/etc
/usr/bin/install -c -m 775 -o nagios -g nagios -d /usr/local/nagios/etc/objects
/usr/bin/install -c -b -m 664 -o nagios -g nagios sample-config/nagios.cfg
/usr/local/nagios/etc/nagios.cfg
/usr/bin/install -c -b -m 664 -o nagios -g nagios sample-config/cgi.cfg
/usr/local/nagios/etc/cgi.cfg
/usr/bin/install -c -b -m 660 -o nagios -g nagios sample-config/resource.cfg
 /usr/local/nagios/etc/resource.cfg
/usr/bin/install -c -b -m 664 -o nagios -g nagios sample-config/template-
object/templates.cfg /usr/local/nagios/etc/objects/templates.cfg
/usr/bin/install -c -b -m 664 -o nagios -g nagios sample-config/template-
object/commands.cfg /usr/local/nagios/etc/objects/commands.cfg
/usr/bin/install -c -b -m 664 -o nagios -g nagios sample-config/template-
object/contacts.cfg /usr/local/nagios/etc/objects/contacts.cfg
/usr/bin/install -c -b -m 664 -o nagios -g nagios sample-config/template-
object/timeperiods.cfg /usr/local/nagios/etc/objects/timeperiods.cfg
/usr/bin/install -c -b -m 664 -o nagios -g nagios sample-config/template-
object/localhost.cfg /usr/local/nagios/etc/objects/localhost.cfg
/usr/bin/install -c -b -m 664 -o nagios -g nagios sample-config/template-
```

```
object/windows.cfg /usr/local/nagios/etc/objects/windows.cfg
/usr/bin/install -c -b -m 664 -o nagios -g nagios sample-config/template-
object/printer.cfg /usr/local/nagios/etc/objects/printer.cfg
/usr/bin/install -c -b -m 664 -o nagios -g nagios sample-config/template-
object/switch.cfg /usr/local/nagios/etc/objects/switch.cfg

*** Config files installed ***
```

(9) 设置目录权限：

```
sudo make install-commandmode
```

输出如下：

```
/usr/bin/install -c -m 775 -o nagios -g nagcmd -d /usr/local/nagios/var/rw
chmod g+s /usr/local/nagios/var/rw

*** External command directory configured ***
```

A.10.2　配置

(1) 配置邮件地址。

(2) 修改联系配置文件：

```
sudo vi /usr/local/nagios/etc/objects/contacts.cfg.
```

将邮件地址`nagios@localhost`改成你自己的。

下面几步配置 Nagios Web 界面：

(3) 把 Nagios Web 配置文件安装到 Apache 的 conf.d 目录：`sudo make install-webconf`

```
/usr/bin/install -c -m 644
sample-config/httpd.conf /etc/apache2/conf.d/nagios.conf
*** Nagios/Apache conf file installed ***
```

(4) 创建登录 Nagios Web 界面的账号：

```
sudo htpasswd -c /usr/local/nagios/etc/htpasswd.users nagiosadmin
```

系统会提示你输入密码并确认。

(5) 重启 Apache：

```
sudo /etc/init.d/apache2 reload
```

A.10.3　编译和安装Nagios插件

前面我们从 www.nagios.org/download/下载了 Nagios 插件，版本是 1.4.15。

编译安装 Nagios 插件的步骤如下。

(1) 解压缩：

```
tar zxvf nagios-plugins-1.4.15.tar.gz
```

(2) 进入插件目录：

```
cd nagios-plugins-1.4.15
```

(3) 配置：

```
./configure --with-nagios-user=nagios --with-nagios-group=nagios
```

(4) 编译：

```
make
```

(5) 安装：

```
sudo make install
```

这样 Nagios 和插件就安装好了，可以启动 Nagios 了。其他配置这里就不再介绍，可以阅读官方文档：www.nagios.org/documentation 了解更多详情。

A.11　RRDtool安装与配置

本节介绍如何在 Linux 和 Unix 上安装 RRDtool。安装 RRDtool 需要 SVN 客户端、automake、autoconf 和 libtool。

源码安装 RRDtool 如下：

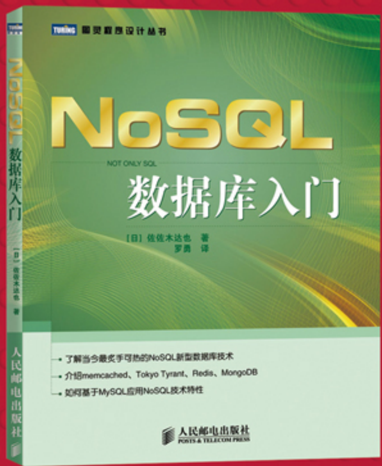
```
svn checkout svn://svn.oetiker.ch/rrdtool/trunk/program
mv program rrdtool-trunk
cd rrdtool-trunk
./autogen.sh
./configure --enable-maintainer-mode
make
sudo make install
```

A.12　MySQL安装Handler Socket

Handler Socket 适用于 MySQL 服务器 5.x 版本。安装如下：

```
git clone git://github.com/ahiguti/HandlerSocket-Plugin-for-MySQL.git
cd HandlerSocket-Plugin-for-MySQL
./autogen.sh
./configure --with-mysql-source=/root/install/mysql-<version number>
--with-mysql-bindir=/usr/bin
make
make install
```

欲了解当今最炙手可热的NoSQL新型数据库技术，下面这本书必读：



本书是一本NoSQL入门书，从最基本的NoSQL发展史开始，介绍了memcached、Tokyo Tyrant、Redis和MongoDB这四种NoSQL数据库的使用背景、优缺点和具体应用实例，并对这四种数据库进行了比较，旨在帮助读者全面了解NoSQL能解决的具体问题，为读者开发数据库提供更多的选择。最后还介绍了如何将MySQL数据库NoSQL化。

深入NoSQL

NoSQL数据库是非常高效、强大的海量数据存储与处理工具。大部分NoSQL数据库都能很好地适应数据增长，并且能灵活适应半结构化数据和稀疏数据集。这本上手指南全面展示了NoSQL数据库的基础概念和实践方案。本书作者、专家Shashank Tiwari首先介绍NoSQL的特点和典型用例，再分析NoSQL适用于应用程序栈的什么位置。他独到的见解能帮助你针对特定的数据存储需求选择最适合的NoSQL方案。

本书主要内容包括：

- 揭示NoSQL数据库的关键概念，包括列族存储、键/值存储以及文档数据库；
- 深入介绍NoSQL产品及Hadoop产品族的安装与配置；
- 解释存储、访问和查询NoSQL数据的方法，使用到的产品包括MongoDB、HBase、Cassandra、Redis、CouchDB、Google App Engine等；
- 检视架构和内部结构；
- 提供最佳实践以及性能调优和扩展配置方面的指导；
- 展示一系列与NoSQL、分布式平台及大规模处理相关的工具，包括Hive、Pig、RRDtool、Nagios等。

- 全面展示NoSQL基础概念和实践方案
- 理解大数据的各种技术架构和思想
- 新潮热门技术一览无余

WILEY
www.wiley.com

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/数据库

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-29638-2
9 787115 296382 >
ISBN 978-7-115-29638-2
定价：69.00元